DOCUMENT RESUME

ED 288 736 SE 048 748

AUTHOR	Farkouh, Alice; And Others
TITLE	Advanced Placement Computer Science (with Pascal). Teacher's Guide. Volume 1. Second Edition.
INSTITUTION	New York City Board of Education, Brooklyn, N.Y. Div. of Curriculum and Instruction.
REPORT NO	CURRIC-00-6090-40; ISBN-88315-571-0
PUB DATE	85
NOTE	237p.; First edition no longer in print.
AVAILABLE FROM	New York City Board of Education, Div. of Curriculum and Instruction, 131 Livingston Street, Room 613, Brooklyn, NY 11201 ($12.00).
PUB TYPE	Guides - Classroom Use - Guides (For Teachers) (052) -- Computer Programs (101)
EDRS PRICE	MF01 Plus Postage. PC Not Available from EDRS.
DESCRIPTORS	*Advanced Placement; College Science; Computer Literacy; *Computer Science; Computer Uses in Education; Course Content; Courseware; Higher Education; *Programing; Programing Languages; *Science Activities; Science Education; Science Instruction; *Secondary School Science
IDENTIFIERS	*PASCAL Programing Language

ABSTRACT

The purpose of this guide is to give teachers and supervisors a working knowledge of various approaches to enhancing pupil learning about computer science, particularly through the use of Pascal. It contains instructional units dealing with: (1) computer components; (2) computer languages; (3) compilers; (4) essential features of a Pascal program; (5) Pascal variables; (6) data types and constants; (7) real and integer data types; (8) character and Boolean data types; (9) standard or built-in functions; (10) IF-THEN statements; (11) IF-THEN-ELSE control structures; (12) nested IF-THEN-ELSE statements; (13) programs involving IF-THEN-ELSE statements; (14) REPEAT-UNTIL structures; (15) WHILE-DO statements; (16) FOR-DO statements; (17) CASE statements; (18) standard files, input and output; (19) Eof and Eoln functions; (20) stepwise refinement; (21) inventory management; (22) functions; (23) local variables and side effects; (24) procedures without parameters; (25) procedures with parameters; (26) parameters in procedures and functions; (27) scope of an identifier; (28) forward references; (29) top-down design; (30) programming methodology; (31) program correctness; (32) recursive functions, part 1; (33) recursive function, part 2; (34) indirect recursion; (35) the Towers of Hanoi problem; (36) iteration vs. recursion; (37) user-defined scalar types; and (38) subrange types. (TW)

****
* Reproductions supplied by EDRS are the best that can be made *
* from the original document. *
****

NEW YORK CITY BOARD OF EDUCATION

DIVISION OF CURRICULUM AND INSTRUCTION
AND THE OFFICE OF TECHNOLOGY

ED288736

ADVANCED PLACEMENT COMPUTER SCIENCE WITH PASCAL

3-5-8-13-21
IF-THEN-ELSE...
RECURSION

U.S. DEPARTMENT OF EDUCATION
Office of Educational Research and Improvement
EDUCATIONAL RESOURCES INFORMATION CENTER (ERIC)
☒ This document has been reproduced as received from the person or organization originating it.
☐ Minor changes have been made to improve reproduction quality.
• Points of view or opinions stated in this document do not necessarily represent official OERI position or policy.

"PERMISSION TO REPRODUCE THIS MATERIAL IN MICROFICHE ONLY HAS BEEN GRANTED BY
Claire Mitchell
TO THE EDUCATIONAL RESOURCES INFORMATION CENTER (ERIC)."

SE 048 748

VOLUME 1

2ND EDITION

BEST COPY AVAILABLE

New York City Board of Education Teacher's Guide to

ADVANCED PLACEMENT

Computer Science

(with Pascal)

VOLUME 1 - SECOND EDITION

New York City Board of Education

James F. Regan, President
Miguel F. Martinez, Vice President

Joseph G. Barkan
Stephen R. Franse
Irene Impellizzeri
Marjorie A. Lewis
Robert F. Wagner, Jr.

Nathan Quinones, *Chancellor*

Charles I. Schonhaut, *Deputy Chancellor*

Jerald Posman, *Deputy Chancellor for Financial Affairs*

Louise Latty, *Chief Executive for Instruction*

Charlotte Frank, *Executive Director*
Division of Curriculum and Instruction

Irwin Kaufman
Director of Technology

Copyright © 1985
by the Board of Education of the City of New York

Application for permission to reprint any section of this material should be made to the Chancellor, 110 Livingston Street, Brooklyn, N.Y. 11201. Reprint of any section of this material shall carry the line, "Reprinted from *New York City Board of Education Teacher's Guide to Advanced Placement Computer Science (with Pascal)* by permission of the Board of Education of the City of New York."

New York City Public Schools should order additional copies of this publication from the Bureau of Supplies.

ISBN No. 88315-571-0 Curric. No. 00-6090-40

Other persons and institutions may obtain copies of this publication from the Curriculum Production Unit of the Office of Curriculum Development and Support, Room 613, 131 Livingston Street, Brooklyn, N.Y. 11201. For information call (718) 596-4902.

NEW YORK CITY BOARD OF EDUCATION
Division of Curriculum and Instruction
Office of Curriculum Development and Support

New York City Board of Education Teacher's Guide to

ADVANCED PLACEMENT
Computer Science
(with Pascal)

VOLUME 1 - SECOND EDITION

It is the policy of the New York City Board of Education not to discriminate on the basis of race, creed, national origin, age, handicapping condition, or sex in its educational programs, activities, and employment policies, as required by law. Inquiries regarding compliance with appropriate laws may be directed to Mercedes Nesfield, Director, Office of Equal Opportunity, 110 Livingston Street, Brooklyn, New York 11201; or to Charles Tejada, Director, Office of Civil Rights, Department of Education, 26 Federal Plaza, Room 33-130, New York, New York 10278.

FOREWORD

In 1983 the Division of Curriculum and Instruction published computer literacy curriculum guides for both elementary and secondary schools. That project was the first step in a comprehensive plan for computer education in New York City. Computer technology is new and it is changing daily. We as educators are faced with a common challenge: how to use this exciting and powerful technology in a way that will meet the changing needs of the students and will continue to accommodate changes as they occur. The challenge is a great one, but the potential for these tools is even greater.

We know that computers can ignite the imagination and extend the limits of human learning. They enable their users to view the world in new ways. When students interact with computers they take charge of their own learning and learn more through this interaction. In short, computers are tools used by humans for human endeavors.

New York City Board of Education Teacher's Guide to Advanced Placement Computer Science (with Pascal) is an important step in the New York City plan for computer education. Pascal is a sophisticated and flexible computer language. As students gain proficiency in Pascal, they develop important concepts in computer science. The content of this advanced placement curriculum will challenge students to master the reasoning skills demanded for programming Pascal. We hope that this guide will aid teachers and students in keeping pace with the challenges of our information age.

Charlotte Frank

Charlotte Frank
Executive Director
Division of Curriculum and Instruction

ACKNOWLEDGMENTS

New York City Board of Education Teacher's Guide to Advanced Placement Computer Science with Pascal, Volume 1 (Second Edition) is a joint project of the Division of Curriculum and Instruction, Charlotte Frank, Executive Director, and the Office of Technology, Irwin Kaufman, Director.

Morris Freedman, Director of the Office of Curriculum Development and Support, provided overall supervision of this project.

Michael Ryan, former Director of the Computer and Information Sciences Unit, gave impetus and guidance to this project by supervising the writing teams, arranging for the planning and production meetings, and screening materials produced by the writers. Joseph Eaione, Assistant Director of the Computer and Information Sciences Unit, provided coordination.

Stephen Radin, Director of Curriculum Development, Office of Technology, served as project coordinator of the second edition.

The writing team was coordinated by Alice Farkouh, Curtis H.S.; Mary Gahn, John Browne H.S.; and David Weinstein of Brooklyn Technical High School.

A team of dedicated professionals wrote the topics and was involved in all aspects of their compilation:

Alice Farkouh, Curtis H.S.
Mary Gahn, John Bowne H.S.
Yvette Beck, Herbert H. Lehman H.S.
Art Seelenfreund, Herbert H. Lehman H.S.
Nick Skouras, John Bowne H.S.
Dave Weinstein, Brooklyn Tech H.S.

Additional advice was provided by James Berlin, Jamaica H.S., Ellen Norden, Francis Lewis H.S., and Bobette Beinhacker, Jamaica H.S.

The following reviewers made helpful comments and suggestions: Phil Bayman, Christopher Columbus H.S.; James DeLuca, Hillcrest H.S.; Neal Ehrenberg, Computer and Information Sciences Unit; Joel Friedberg, Herbert Lehman H.S.; Melvin Klein, Brooklyn Technical H.S.; Marc Licht, Director of the Computer and Information Sciences Unit; Ellen Norden, Bronx H.S. of Science; Stephen Radin, Computer and Information Sciences Unit; Richard Rothenberg, Stuyvesant H.S.; Mark Saul, Bronx H.S. of Science; Elayne Schulman, Staten Island Academy and Arnold Scheiman, Park West H.S.

The manuscript was prepared for publication by the Editorial Unit, under the direction of Alaire Mitchell, Assistant Director of Curriculum Research (Acting):

Editing and production coordination: Ann E. Wolf, Susan Grathwohl, and Vincent Puglia
Word processing supervision: Sarah Corbin
Graphics supervision: Maureen Maquire
Word processing: Regina Paleski, Pat DePaolo, and Janet Taylor
Cover: Victor Roseman

INTRODUCTION

The use of computers in the classroom has become more prevalent during the past few years. Through this use we have developed a knowledge of their functions and a better vision of their potential. We encourage our colleagues to freely experiment and attempt new approaches. Only with all of us working in concert, can we fulfill our vision for an effective integration of school technology into all subject areas. From these beginnings, we will nuture the critical thinking and practical skills our students will need to succeed in the information age. If children are not comfortable with computers, they will be as unprepared for the future as an adult who cannot read or write.

The purpose of this guide is to give teachers and supervisors a working knowledge of various approaches to enhance pupil learning. Our guides are intended to be open ended and flexible in nature. As a 1920 car is not efficient by 1980 standards, outmoded computer education ideas of the seventies have been replaced by the new visions of the eighties. Classroom use of technology has doubled and redoubled in the past few years. Teachers are continually devising new ideas for the use of this technology.

Our primary responsibility in this area and our reason for writing this book, is to see that computers are used as an efficient classroom tool. Only through a real sharing can we move forward in this rapidly changing field of technology. Feel free to write to me directly with your comments and suggestions concerning this curriculum guide, so that we may improve and update it to meet your needs.

Irwin Kaufman

Irwin Kaufman
Director
Office of Technology

TABLE OF CONTENTS

Topic 1
Computer Components 1

Topic 2
Computer Languages 5

Topic 3
Compilers 8

Topic 4
Essential Features of a Pascal Program 10

Topic 5
Pascal Variables 14

Topic 6
Data Types and Constants 19

Topic 7
Real and Integer Data Types 22

Topic 8
Character and Boolean Data Types 27

Topic 9
Standard or Built-in Functions 32

Topic 10
IF-THEN Statements 37

Topic 11
IF-THEN-ELSE Control Structures 42

Topic 12
Nested IF-THEN-ELSE Statements 48

Topic 13
Programs Involving IF-THEN-ELSE Statements 54

Topic 14
REPEAT-UNTIL Structures 60

Topic 15
WHILE-DO Statements 63

Topic 16
FOR-DO Statements 67

Topic 17
CASE Statements 71

Topic 18
Standard Files: Input and Output 78

Topic 19
Eof and Eoln Functions 84

Topic 20
Stepwise Refinement 90

Topic 21
Inventory Management 94

Topic 22
Functions 98

Topic 23
Local Variables and Side Effects 104

Topic 24
Procedures without Parameters 109

Topic 25
Procedures with Parameters 113

Topic 26
Parameters in Procedures and Functions 118

Topic 27
Scope of an Identifier 121

Topic 28
Forward References 126

Topic 29
Top-Down Design 130

Topic 30
Programming Methodology 136

Topic 31
Program Correctness 139

Topic 32
Recursive Functions: Part 1 144

Topic 33
Recursive Functions: Part 2 148

Topic 34
Indirect Recursion 152

Topic 35
Towers of Hanoi Problem 156

Topic 36
Iteration vs. Recursion 160

Topic 37
User-defined Scalar Types 165

Topic 38
Subrange Types 169

Appendix A
Reserved Words 174

Appendix B
Standard Constants, Files, and Types 175

Appendix C
Standard Functions 176

Appendix D
Differences Between Standard Pascal and UCSD Pascal 177

Appendix E
Operator Precedence 179

Appendix F
Operating Systems 180

Appendix G
Scope and Sequence 205

Evaluation Questionnaire 225

Topic 1

COMPUTER COMPONENTS

Performance Objectives: Students will be able to:

- identify the four major components of the computer: CPU, memory (main and secondary), input devices, output devices
- explain the functions of each of the major components
- differentiate between RAM and ROM

Vocabulary: CPU (central processing unit), control unit, ALU (arithmetic logic unit), registers, memory (main and secondary), input and output devices (CRT, keyboard, printer, plotter, card reader), RAM, ROM, data

Motivation:

- If you were an inventory clerk, what would you do if you received a request for a specific item that was not on display?
  - The inventory clerk could go into the stock room and check to see if the item was in stock.
- If the item was not in stock, how would the clerk check to see if it was available in another store?
  - The clerk would call the other store and request a similar check in that store.
- If you worked for a company that had a computer, what process would you go through to look for the item?
  - Enter the part or number on the keyboard.
- How would the computer respond to this request?
  - The computer would search its files and respond with the stores that have the item and the quantities held in inventory.
- How would you receive this information from the computer?
  - It would appear on the screen (cathode ray tube, or CRT).
- Let's look at this process again to understand and examine the four primary components of the computer which allow you to process this information.

Aim: What are the primary components of a computer?

Development:

1. Information (data) can be entered into the computer by many different devices. All of these devices are called input devices. An input device is any device that allows the entry of data into the computer.

13

2. What are some input devices?

   - keyboard
   - keypunch machine and punched cards: a keyboard-like device which punches data onto punched cards (cardboard rectangular cards)

3. How does the information from the punched card get into the computer?

   - A card reader (which is an input device) reads the data from a punched card by light or brush.

4. What happens to the data that entered the computer through an input device?

   - When information enters the computer through an input device, it enters the central processing unit (CPU) which is responsible for controlling all the activities of the computer.

5. What is contained in the CPU?

   - The CPU is composed of three parts:

     control unit
     arithmetic logic unit (ALU)
     various registers

6. The control unit is the brain of the CPU. It coordinates all the activities and actions of the computer. When a program (a series of instructions) is to be executed, the control unit oversees the transferring of data to and from the memory and the ALU. In addition, the control unit will identify input/output operations and will direct the flow of data accordingly.

7. What happens to the input data and/or programs that enter the CPU?

   - The data and programs that have been input into the computer are stored in main memory or the primary storage area. The main memory can be depicted as mailboxes in a post office with distinct addresses. Each of these mailboxes, or memory cells, can contain information, with the restriction that the mailbox may contain only one piece of information at a time. If any new information is placed in that cell, the previously stored information is erased from memory and cannot be retrieved.

8. The primary storage area is divided into two types of memory:

   - RAM
   - ROM

   . RAM (Random Access Memory) is the part of the memory that is used to store user created data and programs. Data and programs can be retrieved from RAM and changed as needed by the program.

. ROM (Read Only Memory) is the part of the memory that cannot be changed. It might contain predefined functions or procedures that are permanently stored in the computer. User programs are never written and stored in ROM.

9. What happens when all the memory in the computer has been used?

   - Additional storage area is provided. This area is called secondary storage. Secondary storage is the additional storage area that is connected to the computer's CPU. Devices used for secondary storage include magnetic tapes and hard and floppy diskettes.

10. How are arithmetic operations and comparisons made within the computer?

    - Arithmetic operations and comparisons are performed in the arithmetic logic unit (ALU). The ALU is the part of the CPU that performs two tasks: arithmetic operations such as addition, subtraction, multiplication, division, etc., and logical operations, operations that compare two values or show the relationship between two values. For example:

      Is A < B
      Is age > = 65
      Is hours = 40

      When a stored program directs the computer to perform arithmetic or logical operations, the CPU activates the ALU.

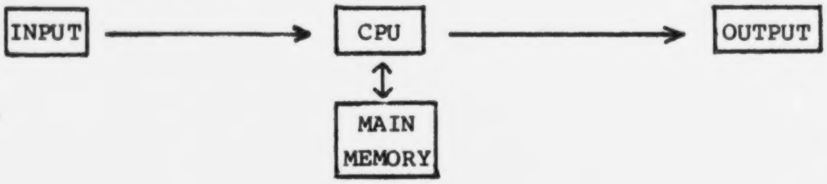
11. Data and information used within the program are stored in temporary storage areas called registers. The registers are temporary holding places for the data until the operations are completed. Once the operations are completed, the desired information can be retrieved from the computer.

12. Output is the retrieval of desired information from the computer. Output can be displayed in the following ways:

    - Screen (CRT): Output displayed on a CRT is also called softcopy.
    - Printer: Output device used to print information on paper, which is called hardcopy.
    - Plotter: Plotters convert the desired output into pictorial diagrams or graphs.

13. What are the four main components of a computer?

    - CPU
    - memory
    - input device(s)
    - output device(s)



Summary:

1. What are the four major components of the computer?
2. Describe the function of these components with respect to the entry of data into the computer.
3. What is the difference between RAM and ROM?

Topic 2

COMPUTER LANGUAGES

Performance Objectives: Students will be able to:

- state the need for a variety of computer languages
- describe the characteristics of some of the more popular languages
- differentiate Pascal from BASIC

Vocabulary: low-level language, high-level language, machine language, assembly language, structured programming, BASIC, Pascal, UCSD Pascal, COBOL, FORTRAN

Motivation:

- These are examples of low-level language commands:

```
001 0001          10001111          000010000          (LD HL, MEMLOC)
```

- The code is written at a level directly discernable by the computer. Each is a computer command to load into register HL the contents of the memory location specified by the variable memloc.
- A more readable language for people, however, might be:

```
LET HL=MEMLOC
```

- Programs written in this English-oriented form are characteristic of high level languages.

Aim: What are the characteristics of some of the higher-level languages?

Development:

1. Computers are made up of circuits that can exist in two states--ON or OFF. For this reason machine language is binary (ones and zeros). Similar to machine language is assembly language, which is based on mnemonics for readability.

2. Low-level languages (e.g., machine language) give the programmer direct control over such details as memory locations and microprocessor registers. The disadvantage is that the programmer has to remember the codes for the various operations as well as keep track of the locations in main memory of all the data.

3. In the second half of the 20th century, the need to communicate with a computer in more general terms was recognized. Higher-level languages were needed which could deal with abstractions such as arrays and variables in symbolism closer to ordinary English.

4. The designers of the new languages had to decide what their goals were. Was the objective to create scientific programs? Business applications? Was their purpose educational?

5. The names of most computer languages mnemonically illustrate their purpose. There are more than 200 computer languages in use today. Below are brief descriptions of a few of the most popular.

   - FORTRAN (FORmula TRANslator): In 1956 this compiled language was developed by IBM for problem solving in the scientific area. It was the first popular high level language.

   - COBOL (COmmon Business Oriented Language): In 1960, this high level language was developed for commercial applications. It was written in phrases easily understood by accountants and businessmen.

   - BASIC (Beginner's All-Purpose Symbolic Instruction Code): In 1964, graduate students at Dartmouth set out to design a high level language that would be quick to learn, easy to write, and could run short programs. BASIC is built into most home computers on the market today. All the other languages must be loaded into memory from external storage devices such as diskettes and cassettes.

   - PL/I (Programming Language/I): In 1964, IBM introduced this general-purpose programming language in the hopes of creating a standard. A committee, comprised of members of SHARE, the largest user's group for IBM mainframes, developed the specifications. PL/I incorporates elements of FORTRAN, COBOL, Algol, and several other languages. PL/C, a subset of PL/I, is extremely popular on college campuses.

   - Pascal: In 1971, Nicklaus Wirth (not Blaise Pascal) developed this compiled, structured language. Pascal was developed to help teach a systematic approach to good program design. By declaring variables according to type at the beginning of the program, potential misuse of variables is reduced. Pascal's versatility is also enhanced by the variety of data types and data structures, and the capability of user-defined data types.

     Wirth's standard Pascal has been modified to provide specialized facilities not available in the original version. One modified version was developed at the University of California at San Diego, and is called UCSD Pascal. Further modifications are common on individual systems, such as Apple Pascal.

     Pascal's advantage over BASIC becomes apparent when writing long and complex programs. The readability due to the modularization inherent in Pascal makes it much easier to understand and debug than a similar program written in BASIC. However, the distinct advantage of BASIC over Pascal is seen in the writing of short, simple programs. BASIC programs can be typed in and run directly. Syntax errors can be corrected easily with a new version running without delay. In Pascal, the entire program must be recompiled before it can be run again.

. Other languages that have their own adherents include: FORTH, APL, LISP, MODULA-2, PILOT, LOGO, SMALLTALK, and PROLOG.

Summary:

1. Why are there so many computer languages?

2. What is the difference between a low level and a high level language?

3. What are the advantages and disadvantages of Pascal versus BASIC?

Exercises:

1. For a language to be useful, it must be standardized to be portable. Look up "portable" in a dictionary and explain how it applies to a computer language.

2. Choose a language, research it, and write a brief summary describing its inception.

Topic 3

COMPILERS

Performance Objectives: Students will be able to:

- explain the difference between a compiler and an interpreter
- describe the compilation process
- explain the difference between compile-time, run-time, and output errors
- identify the processing stage where errors are detected

Vocabulary: compiler, interpreter, compile-time error, run-time error, output error, syntax error, source code, object code, machine language, object program, source program

Motivation:

- What language does computer hardware understand?

Aim: How does a computer "understand" different high level languages?

Development:

1. A computer's circuits understand only binary numbers. E.g., 10110101 10001010 might mean "add." A program written as a series of binary numbers (zeroes and ones) is called a machine language program or an object program or object code.

2. How does a computer understand languages like Pascal?

   - Programs written in high level languages like Pascal cannot be directly obeyed by the computer's circuits. They must first be translated into machine language. This is done by a program called a compiler, the process being called compilation. The high level language program that is translated by the compiler is called the source program or source code.

3. What are compile-time errors?

   - The compiler looks at the source program it is given and checks it to see that it follows the various syntactic and semantic rules of the language. If it finds violations of these rules, diagnostic messages (error messages) are output. These are called compile-time errors because they are found at compilation time. If no such errors are found, the compiler produces an object program which is a translation of the source program with which the compiler started. Errors in a compiled program are difficult to trace and correct.

3. How is it that most microcomputers seem to automatically understand BASIC?

   - In some systems (only for certain simple languages like BASIC) the compilation process can be avoided. A program called an interpreter looks directly at the source code, translates it one line at a time, and stops when an error is found.

4. Programs run by interpreters generally run significantly more slowly than compiled programs. This is because an interpreter translates each line of code each time it is executed (as in a loop). A compiler translates the entire program only once. However, interpreters tend to give clearer diagnostic messages and avoid the compilation process, so there is some balance. A compiled program executes on the average about ten times faster than an interpreted program.

5. (Optional) There can be an intermediate system where the source code is compiled into the object code of a hypothetical computer. The UCSD "P" system is an example of this. The program is run by a computer using a program that simulates the hypothetical one. The advantage of this is that the entire system is portable, meaning all that has to be done to make the system work on a new real computer is write the program to allow the real computer to simulate the hypothetical one. Thus, UCSD Pascal can run on both the Apple IIe and the IBM PC despite the fact that they have entirely different processors.

6. What are run-time errors?

   - When a program is run, additional errors may be discovered; e.g., division by zero, taking the logarithm of a negative number. Such errors are called run-time errors.

7. It is also possible for a program to compile successfully, look like a valid Pascal program, have no run-time errors, and still be wrong. The program doesn't produce the output you expect when test data (giving known results) are input to the program. This is known as an output error. The program doesn't do what it was written to do. Only well planned test data can uncover such bugs. The source of the error is, most likely, the algorithm.

Summary:

1. Compare and contrast compilers and interpreters.
2. Name three different types of programming errors.
3. Tell how the types of errors differ from one another.

Exercises:

1. Give an example of each of the following types of errors: syntax, semantic, output.

Topic 4

ESSENTIAL FEATURES OF A PASCAL PROGRAM

Performance Objectives: Students will be able to:

- recognize essential features of all Pascal programs (program heading, BEGIN and END)
- state the difference between reserved words and identifiers
- use a simple write or writeln statement in a program

Vocabulary: identifier, standard identifier, reserved word, program heading, compound statement, PROGRAM, BEGIN, END, write, writeln

Motivation:

- What will the output of the following program?

```
PROGRAM First (output);
BEGIN
  writeln ('This is my first program.')
END. (* First *)
```

Aim: What are the mandatory features of a simple Pascal program?

Development:

1. Every Pascal program has a program heading with this general form:

```
PROGRAM      __________________         (input, output);
             program identifier
```

A Pascal program heading must start with the reserved word PROGRAM followed by a program name that is used to identify the program. In addition, the program heading must specify in parentheses the type of file that will be needed by the program. If a program calls for inpu only, that is, reading in data, the input file must be specified:

```
PROGRAM  __________________  (input);
         program identifier
```

If a program is used to produce output only, the output file must be specified:

```
PROGRAM  __________________  (output);
         program identifier
```

When a program uses both input and output, both must be specified:

22

```
PROGRAM ____________________ (input, output);
          program identifier
```

A semicolon (;) is used at the end of the heading to separate the program heading from the next statement.

2. What is a reserved word?

   - A reserved word is a word that has a special meaning to the Pascal compiler. You cannot use the reserved word as an identifier or assign another meaning to it. It must only be used for the specific purpose it was developed for. PROGRAM, BEGIN, END, VAR, CONST, IF, THEN, ELSE, OR are reserved words. For example, can you name a program "program" and use this heading?

     ```
     PROGRAM Program (input, output);
     ```

     No, since PROGRAM is a reserved word, it cannot be used to identify the name of a program.

3. What is an identifier?

   - An identifier is a name that is used to refer to a variable in a program, procedure, function, or parameter. Identifiers are chosen by the programmer. These are the rules for writing Pascal identifiers:

     (a) Identifiers must always start with a letter. For example, Grade is a legal identifier. 1 grade is illegal.
     (b) After the first character you may have any combination of letters and/or digits.
     (c) No punctuation or spaces are permitted. For example, M Mouse is not a legal identifier.
     (d) An identifier may not be a reserved word.

     An identifier may contain more than eight characters (letters or numbers). However, only the first eight characters will be examined by the compiler. For example, these two programs will not be stored differently because the first eight letters of the program names are the same:

     ```
     PROGRAM Greeting1  (input, output);
     PROGRAM Greeting2  (input, output);
     ```

4. What is a standard identifier?

   - A standard identifier is a word that has a special meaning to the compiler but can be redefined by the programmer, e.g., write, readln, maxint (although this is not recommended).

5. What does a program contain?

   - Programs must contain a program heading and statements for the computer to execute.

6. The statement part of the program contains the statements that the computer will execute. Statements are "sandwiched" between the reserved words BEGIN and END. BEGIN and END are delimiters. For example:

```
BEGIN
  statement 1;
  statement 2;
  ...
  statement n
END.
```

   The group of statements sandwiched between BEGIN and END is a compound statement. These are the rules for writing statements:

   (a) Statements are separated by semicolons.
   (b) There is no semicolon between BEGIN and the first statement, nor between the last statement and END.
   (c) For every BEGIN there must be an END.
   (d) The period following the END indicates the termination of a program.

7. What commands can we use to produce output?

   - In Pascal, the commands used to produce output are write and writeln (pronounced writeline). Write will cause the computer to print the output in succession (on the same line). Writeln will cause the computer to print the specified output and then move to the next new line.

8. How can comments be incorporated into a program?

   - Comments are enclosed in braces { } or (* *) and may be inserted anywhere in a program where a blank would normally occur.

Applications

1. Which are legal identifiers?

   (a) Cost   (d) 3CPO
   (b) B9   (e) R2D2
   (c) As   (f) Gross Pay

2. Write the outputs for each of the following:

(a)

```
write (1, 2, 3, 4, 5);
```

Answer:

```
1  2  3  4  5
```

(b)

```
write (1, 2);
write (3);
write (4, 5);
```

Answer:

```
1  2  3  4  5
```

(c)

```
writeln (1, 2);
writeln (3);
writeln (4, 5);
```

Answer:

```
1    2
3
4    5
```

(d)

```
write (1);
writeln (2);
writeln (3);
writeln (4, 5);
```

Answer:

```
1    2
3
4    5
```

3. Are write and writeln standard identifiers? Why?

4. Write a program using write and writeln to produce the following output:

```
1    2
3
4    5
```

Possible solution:

```
PROGRAM Numbers (output);
BEGIN
  writeln (1, 2);
  writeln (3);
  writeln (4, 5)
END. (* Numbers *)
```

Summary:

1. What are the main features of a Pascal program?
2. Differentiate among identifiers, standard identifiers, and reserved words.
3. What is the difference between write and writeln?
4. How can a line in a program be skipped?
5. Write a program that would produce a skipped line in the output.

Topic 5

PASCAL VARIABLES

Performance Objectives: Students will be able to:

- give the name, type, and value of a variable
- input information interactively using the read and readln statements with prompt messages
- write simple variable declarations and assignment statements

Vocabulary: variable declaration, assignment statement, variable, type

Motivation:

- Write a program to evaluate $5^2 + 12^2$. (Elicit responses.)

- This is one solution to the problem:

```
PROGRAM Pythag (output);
BEGIN
  writeln (5 * 5 + 12 * 12)
END. (* Pythag *)
```

- What if we now wanted to take the square root of this answer without recalculating $5^2 + 12^2$?

  - We would have to store the answer in a variable.

Aim: How do we assign values to variables in a Pascal program?

Development:

1. In BASIC, how did we assign the value of 5 to X?

   - LET X = 5

2. In Pascal the assignment symbol is

   `:=`

   We read this as "colon equals." For example:

   `X := 5`

3. How would we assign $5^2 + 12^2$ to a Pascal variable? Let's use a meaningful variable identifier:

   `sumofsquares := 5 * 5 + 12 * 12`

To make efficient use of memory space and for readability, we declare the variables we will be using and the type of data they will hold. For example, by declaring X to be an integer, less space has to be reserved in memory than for a real variable. For a real variable, memory reserves room for the integral as well as the decimal part of the number.

A full discussion of data types will follow this lesson.

4. The variables must be declared in a variable declaration block before they can be used in a program:

```
VAR
  X,Y : real;
  ch : char;
  W : integer;
  really : boolean;
```

5. This is the general form of the <u>variable declaration:</u>

```
VAR
  identifier : type;
```

Note the colon separating the identifier from the type and the semicolon following each type. If you have more than one identifier of the same type, separate them with commas.

6. The variable declaration block follows the program header:

```
PROGRAM Arty;
VAR
  cost : real;
```

7. In the main body of the program we may make assignments to the variables we have just declared. We end the assignment statement with a semicolon. For example:

```
cost := 12.50;
```

8. In a real constant, there must be at least one digit before and after the decimal point. For example, if R1 and R2 are reals, then:

```
R1 := 5.0;     R2 := 0.08;
```

There may be expressions on the right side of the

```
:=
```

If

```
L := 8 * 3 + 1
```

then 25 will be assigned to L. The expression is evaluated and the assignment made. The assignment

```
area := base * height
```

is legal if base and height were previously assigned values of the same type as area and declared in the variable declaration block.

9. How would we assign a value to a variable from the keyboard during the run of a program?

   - In BASIC, we write <u>INPUT N</u>. In Pascal, we use <u>read (N)</u>.
   - The identifier must be enclosed in parentheses.

10. What does the following program do?

```
PROGRAM Whatdo (input, output);
VAR
  number : integer;
BEGIN
  writeln ('Type in an integer  ');
  readln (number);
  write ('Today your lucky number is  ');
  writeln (number)
END. (* Whatdo *)
```

The program asks for and receives a number on input from the keyboard. The screen will display on one line the string of characters followed by the number that was read in.

11. Prompt messages should always precede read statements. Whereas one read statement can be used to input several variables. Following the statement

```
read (A, B, C)
```

the computer will await input. The operator should type in three values separated by spaces, not commas, to effect the respective assignments.

12. A variation of read is readln. The standard identifier readln is a built-in function that will assign values to the variables in parentheses. It will discard any extra values on that line. Subsequent read or readln statements will take their data from the next line of input data. This is in contrast to a read statement that will use the data remaining on the line should further read or readln statements be encountered.

13. Consider the statements:

```
readln (A, B, C);
readln (D);
```

If the data line was:

```
1    2    3    4
5    6
```

The following assignments would be made:

1 → A   2 → B   3 → C   5 → D

Insufficient data, however, is a serious problem. The program will not continue if it is waiting to read another value that has not been keyed in.

14. Whereas numeric data must be separated by a space, character input should be contiguous. In response to

```
read (A, B, C, D);
```

the input of

```
*!$;
```

would cause these assignments:

* → A   ! → B $ → C   ; → D

Summary:

1. How does readln differ from read?
2. What is the syntax of a variable declaration block?
3. What symbols are used in making an assignment to a variable in a Pascal statement?

Applications:

1. Match each value below with its corresponding data type:

VALUES	TYPE OF DATA
_____ decimal numbers	(a) boolean
_____ whole numbers	(b) character
_____ true and false	(c) integer
_____ single letters	(d) real

Answers: d, c, a, b

2. Write the assignment statements necessary to assign:

   100 → A   3.5 → B   TRUE → C   'N' → ans

   Answers: A := 100;   B := 3.5;   C := TRUE;   ans := 'N';

3. Fill in the missing types in the variable declaration block below:

```
VAR
  aptnumber     :         ;
  hrlyrate      :         ;
  firstinitial  :         ;
  outtolunch    :         ;
```

   Answers: integer, real, char, boolean

4. Write the Pascal statements that would make the following assignments:

   .25 → dec   10 → wh   'Y' → ans

   Answers: dec := 0.25;   wh := 10;   ans := 'Y';

5. For the input lines below, give the values of A, B, C, D, and E.

```
read (A, B);
readln (C);
readln (D);
readln (E);
```

```
6     5     4     3
2     9
8
```

   Answers: 6 → A   5 → B   4 → C   2 → D   8 → E

Topic 6
DATA TYPES AND CONSTANTS

Performance Objectives: Students will be able to:

- list and describe the four predefined data types
- state the difference between constants, variables, and literals
- explain the benefits of using constant declarations
- write a constant declaration
- find the largest and smallest possible integer values that can be used for a given computer

Vocabulary: integer, real, char, boolean, CONST, maxint

Motivation:

- Write a program that will input a number, n, and output on four separate lines $n\pi$, π/n, n/π, and $n + \pi$. (Use $\pi = 3.14159$.)

- This is one solution to the problem:

```
PROGRAM Usingpi (input, output);
VAR
  number : real;
BEGIN
  writeln ('Enter a number');
  readln (number);
  writeln (number * 3.14159);
  writeln (3.14159 / number);
  writeln (number / 3.14159);
  writeln (number + 3.14159)
END.  (* Usingpi *)
```

Aim: How are Pascal constants used?

Note: Four predefined data types were examined in Topic 5. Your coverage of these data types in this lesson will depend on the depth of coverage in Topic 5.

Development:

1. As we saw in the previous lesson, a variable can take on any value of a certain type. This value may change during the execution of the program.

   - A literal has only one value, and its type is stated implicitly. For example: 2.3 is a real literal, 7 is an integer literal, A is a char literal, and true is a boolean literal.

   - A constant, like a variable, gets assigned a value and a name. The difference between a constant and a variable is that a constant gets a value (and implicitly a type) assigned at the beginning of a program; once assigned, this value cannot be changed. Thereafter, a constant is like a literal that has been given a name.

2. The general form of a constant declaration is:

```
CONST
  constantname1 = constantvalue1;
  constantname2 = constantvalue2;
  ...
  constantnamen = constantvaluen;
```

3. Constant declarations are made before variable declarations.

4. There are two advantages to using constant declarations: programs are easier to read if the value of a constant is replaced by a more meaningful name (e.g., pi for 3.14159), and programs are easier to modify if we can change a constant once rather than look for every occurence of it in the program. In exercise 3 below, for example, we can change all values of the tax rate by just changing the value of the constant taxrate.

5. The largest integer value depends on the computer used, but it has a constant value called maxint. This value is predefined by the Pascal system. The smallest integer value is -maxint.

Summary:

1. What is a literal?

2. What is a constant?

3. What is a variable?

4. Why are constants used?

5. What are the largest and smallest possible integer values?

Exercises:

1. Write a program that will print out the largest and smallest integer values for your computer.

   Possible solution:

```
PROGRAM Printvalues (output);
BEGIN
  writeln ('Largest integer =  ', maxint, '.');
  writeln ('Smallest integer =  ', -maxint, '.')
END.  (* Printvalues *)
```

2. Rewrite the program Usingpi with the constant pi = 3.14159.

   Possible solution:

```
PROGRAM Usingconstantpi (input, output);
CONST
  pi = 3.14159;
VAR
  number : real;
BEGIN
  writeln ('Enter a number');
  readln (number);
  writeln (number * pi);
  writeln (pi / number);
  writeln (number / pi);
  writeln (number + pi)
END.  (* Using constantpi *)
```

3. Using a taxrate of 0.0825, write a program that will input a money value, and output the money value, tax on the money value, and the total cost of money value plus tax. (Note: tax = moneyvalue * taxrate.)

   Possible solution:

```
PROGRAM Taxcalculation (input, output);
CONST
  taxrate = 0.0825;
VAR
  moneyvalue, tax, total : real;
BEGIN
  writeln ('Enter your money value');
  readln (moneyvalue);
  tax := moneyvalue * taxrate;
  total := moneyvalue + moneyvalue * taxrate;
  writeln ('Moneyvalue is ', moneyvalue, '.');
  writeln ('Tax is ', tax, '.');
  writeln ('Total is ', total, '.')
END.  (* Taxcalculation *)
```

Topic 7
REAL AND INTEGER DATA TYPES

Performance Objectives: Students will be able to:

- evaluate integer and real expressions using the precedence rules for operators
- explain the terms range, precision, and exponential notation for real numbers
- state when implicit type conversion occurs between integers and reals
- indicate how the writeln statement may be used with attributes
- discuss the limitation of precision for reals, and its consequences

Vocabulary: range, precision, exponential notation, floating point number, truncation, overflow, underflow, +, -, *, /, DIV, MOD, real, write (number : fieldwidth), write (number : fieldwidth : precision)

Motivation:

- What would be printed by these Pascal statements? (Elicit responses.)

```
(a) writeln (187, 16 + 50, 7 * 41, 3 - 8, 25 DIV 5, 24 MOD 5)
(b) writeln (maxint + 1)
```

- Notice that in statement a, several operations must be performed on integer values.

- Inform students that there are five different arithmetic operators that may be applied to literals, constants, and variables of type integer:

```
+   -   *   DIV   MOD        ( "/" is used for division of reals. )
```

- The first three operators represent the familiar operations of addition, subtraction, and multiplication.

- When A and B are of type integer:

  `A DIV B`

  yields the integral part of the result when A is divided by B. The process of disregarding the fractional part is called truncation.

- The expression

  `A MOD B`

  yields the remainder when A is divided by B. For example:

  (a) 13 + 5 (= 18)
  (b) 7 * 41 (= 287)
  (c) 3 - 8 (= -5)
  (d) 18 DIV 2 (= 9)
  (e) 34 DIV 5 (= 6)
  (f) 2 DIV 3 (= 0)
  (g) 17 MOD 3 (= 2)
  (h) 21 MOD 7 (= 0)
  (i) 3 MOD 8 (= 3)

- We can now see that writeln statement a will print the values 187, 66, 287, 5, and 4. The computer prints these values right-justified in a default fieldwidth that is dependent on the particular system used. (You will need to find the default value for your system.) If the default value is 10, then the integers will be printed as:

```
_______187________66_______287_________5_________4
```

- The default value may be overridden by the listed statement with attributes.

```
writeln (inum : fieldwidth)
```

where inum and fieldwidth are integer expressions. If the fieldwidth is too small to contain inum, then the number inum will be printed correctly in a field of the smallest possible width. Therefore, a fieldwidth of 1 prints a number in a field with no blank spaces. For example:

```
writeln (8 - 30 : 6, 6 * 40 : 4, 270 DIV 10 : 1)
```

would print:

```
   -22 24027
```

The statement

```
writeln (maxint + 1)
```

would produce an overflow error, since maxint is the largest integer value that can be represented by the computer. Because much larger numbers are needed for most scientific and business applications, real numbers must be used.

Aim: What are the differences between real and integer data types?

Development:

1. In Pascal, a real number has a decimal point, at least one digit to the left of the decimal point, and at least one digit to the right of it. For example:

valid real numbers:	2.78		-317.0
invalid real numbers:	17.	.8	45

2. Another representation for reals, called exponential or scientific notation, has been provided for dealing with the very small and very large numbers that are encountered in scientific applications. In this representation the symbol E is understood as "times ten to the power of." For example:

Distance from Earth to Sun (in miles) 93,000,000 = 9.3×10^7 = 9.3E+7

Gravitational constant .0000000000667 = 6.67×10^{-11} = 6.67E-11

Since a number such as 27.3 can be expressed as .273E+2, 2.73E+1, 273E0, and so on, we often refer to reals written in exponential notation as floating point numbers. (The decimal value before E is called the mantissa, and the integer after E is called the exponent). This syntax diagram shows all the possible forms of real numbers:

real ± digits.digits E ± digits

3. Range and precision are two important characteristics to consider for real numbers. Range refers to the set of permissible values, while precision refers to the number of significant digits. Both characteristics are implementation dependent. For example, the range of a particular computer can be 10±38 and its precision can be 6 decimal places:

-10^{38} -10^{-38} 0 10^{-38} 10^{38}

Very large numbers would be out of range and give overflow errors. Very small numbers would be out of range and give underflow errors. Due to the limited precision, 25600.214 would be stored as 25600.2. Note that -10^{38} is small in the sense of being negative, while 10^{-38} is small in absolute value.

4. There are four different operations that can be applied to a real value:

+ - * /

Each operation is interpreted in the customary way. Division (/) always yields a real result. In general, the result of evaluating an expression that contains both reals and integers will be of type real because integers are a subset of type real. Thus, "/" is said to yield a real result.

5. There are precedence levels for evaluating real and integer expressions:

Level 1: * / DIV MOD
Level 2: + -

Level 1 operators come before level 2 operators, while operators at the same level are evaluated from left to right. For example:

(a) 7.0/2 (= 3.5)
(b) 1/3 (= .333333)
(c) 3.2 + 5.0/4.0 =
3.2 + 1.25 (= 4.45)
(d) 2 * 3.8 + 17 MOD 3 =
7.6 + 17 MOD 3 =
7.6 + 2 (= 9.6)

6. Although it is not possible to assign real values to an integer variable, it is possible to assign integer values to a real variable. When this latter type of assignment is executed, the integer value of the expression on the right is converted to a real value. (This can always occur, since the range of reals is greater than the range of integers.) If possible, avoid mixed assignments and mixed expressions.

7. Let's look at the write statement for printing real values. If <u>rnum</u> is of type real,

```
write (rnum : fieldwidth : precision)
```

   will print the number <u>rnum</u> in a field of "fieldwidth" spaces with "precision" digits to the right of the decimal point. If the precision is omitted, exponential notation is used. The fieldwidth must be large enough to allow for the sign of the number.

8. Due to the limited precision with which reals may be represented, operations on reals do not always yield exact values. For example, 1/3 x 3 = 1 in a computer, since .333333 x 3 = .999999 = 1. The slight inaccuracy can have a very damaging cumulative effect when a repeated sequence of operations is performed. To avoid errors when dealing with reals:

   - Never test real values for equality. Instead, check whether the difference between them is less than some small preassigned value. For example, suppose we have the declarations:

```
CONST                               VAR
  epsilon = 1E-6;                     X, Y: real;
```

     To check whether X and Y are equal, do not test to see whether X = Y; instead test whether abs (x-y) < epsilon.

   - Monitor intermediate results in a program that requires a lengthy sequence of calculations.

   <u>Note:</u> IN UCSD Pascal, a <u>long integer</u> type is implemented that enables the programmer to use integers of up to 36 decimal digits.

<u>Summary</u>:

1. What are the precedence rules for operators using integer expressions? real expressions?

2. What are the definitions for the terms <u>range</u>, <u>precision</u>, and <u>exponential notation</u>?

3. What are the limits of precision for reals? What are the consequences of these limits?

4. How can the writeln statement be used with attributes?

Exercises:

Complete these problems and test them on a computer:

1. Evaluate the following expressions:

    (a) `4 / 5`
    (b) `4 DIV 5`
    (c) `17 MOD 3`
    (d) `12 MOD 20`
    (e) `3 DIV 2 * 4 - 5 * 6`
    (f) `3 + (18 - 5) MOD 4 * (2 * 3)`

2. Write the output for each statement. Assume the default fieldwidth is 10 and the default precision is 6.

    (a) `write (8 * 7 : 5, 6.7 : 6 : 3, 17 * 10 : 1)`
    (b) `writeln (6 MOD 8 : 3, 2 / 3, 16804)`
    (c) `writeln (7. 32 : 8, 7.32 : 8 : 2)`

3. Write a program that reads in three test scores and computes and prints their average.

Topic 8
CHARACTER AND BOOLEAN DATA TYPES

Performance Objectives: Students will be able to:

- give examples of appropriate values for each data type
- explain what is meant by ASCII
- evaluate boolean and relational expressions
- explain what flags are

Vocabulary: boolean expression, relational operator, ASCII, char, boolean, AND, OR, NOT, <, >, =, <>, <=, >=, TRUE, FALSE

Motivation:

- If you could only use numbers, how could you transmit an alphabetical message? (Elicit responses.)

- One solution would be to develop a code, for example:

    A = 1
    B = 2
    C = 3
    etc.

Aim: How do we work with character and boolean data types in Pascal?

Development:

1. What are characters?

    - Characters are all of the various symbols that are input and output by a computer: letters, numbers, punctuation marks, etc.

2. How does a computer work with characters?

    - There is a standard code called ASCII which is used by most computers. ASCII stands for American Standard Code for Information Interchange; it is an 8 bit binary code.

3. There are 256 (2 to the 8th power) possible characters that can be represented.

4. What are the rules that govern the use of the character type?

    - The program segment below illustrates how this is done:

```
CONST
  this = 'M';
  that = 'Q';
VAR
  B, O : char
BEGIN
  B := 'I';
  C := ''''; (* This is how you represent a single apostrophe *)
  writeln (B, C, this)
(* This program segment would print I'M *)
END.
```

5. The character variable and constant may not exceed one character in standard Pascal.

6. What does boolean mean?

   - The boolean data type is named after the mathematician George Boole, who invented a mathematics in which the quantities dealt with have only two values, for example, 1 and 0, or TRUE and FALSE. In Pascal, boolean variables can only be assigned the values TRUE or FALSE.

7. How do we declare boolean variables?

```
VAR
  A, B : boolean;
```

   Note: Boolean variables cannot be input, but they can be printed sometimes.

8. There are three boolean operators: AND, OR, NOT

		A	
	AND	0	1
B	0	0	0
	1	0	1

		A	
	OR	0	1
B	0	0	1
	1	1	1

A	
A	NOT A
0	1
1	0

9. Relational operators are used to compare numbers. The <u>relational operators</u> are:

=	equals
<>	not equal
>	greater than
<	less than
>=	greater than or equal to
<=	less than or equal to

10. Expressions using relational operators yield boolean values.

11. Notice that

```
=
```

is a relational operator while

```
:=
```

is the assignment operator. They are not interchangeable.

12. Boolean operators take precedence over relational operators. (See Appendix E for a complete listing of operator precedence.)

Note: This is totally different from most other computer languages such as BASIC, FORTRAN, COBOL, PL/I, ALGOL, etc.

13. This order of operations implies that you need parentheses around the quantities you are comparing, for example:

```
3 > 2 AND 5 = 3 + 1
```

would be valid in most other languages, but will give you an error in Pascal. It must be written:

```
(3 > 2) AND (5 = 3 + 1)
```

14. Let's evaluate that expression:

```
(3 >2) AND (5 = 3 + 1)
  T    AND     F
        F
```

15. It is sometimes necessary to test something early in a program but not make use of the information until later in the program when the values to be tested will no longer be available for testing. Such a test may be made using a boolean variable. The program segment below illustrates this concept:

```
VAR
  young : boolean;
  age : real;
BEGIN
  writeln ('How old are you?');
  readln (age);
  young := age <= 10.0;
...(* much later *)
  (* Don't write cute messages to users unless they are young *)
  IF young THEN writeln ('My, my what a fine job you are doing!!!');
END.
```

Note: The boolean variable young works in the following manner: IF age is less than or equal to 10, THEN young is assigned the value true. IF age is greater than 10, young is assigned the value false. When the variable young is used to signal a choice between two courses of action, it is commonly referred to as a flag.

The statement IF young THEN .... means: IF young is true THEN ....

16. In Pascal there are additional forms that an assignment to a boolean variable may take:

```
(a) test := true;
(b) data := false;
(c) flag := I = J;
(d) finished := test AND data;
(e) goodinput := flag OR (M < N)
```

- (a) and (b) are straight forward.
- In (c), when I = J, flag will take on the value TRUE; otherwise if I ≠ J, flag will take on the value FALSE.
- In (d), since AND requires that both conditions be true for the conjunction to be true, finished will be true only if test is true and data is true.
- In (e), goodinput will be true when at least one of the two conditions is true. If flag is false and M > N, then goodinput will be assigned the value false.

Summary:

1. What are the definitions and Pascal symbols for the relational operators?
2. What does the acronym ASCII stand for?
3. How does a computer work with characters?
4. Can boolean variables be inputted? printed?
5. What are the definitions for the three boolean operators AND, OR and NOT?
6. What is the order of operations for boolean and relational operators?
7. What is the difference between a relational operator and an assignment operator?

Exercises:

1. Given the following values for boolean variables A, B, C, and D, evaluate the boolean expressions.

   A = FALSE   B = TRUE   C = TRUE   D = FALSE

   (a) (A AND B) AND (C AND D)
   (b) (A AND B) OR (B AND C)
   (c) NOT (A AND D) OR (A AND B)
   (d) (A AND B) NOT C AND D AND A
   (e) (B OR C) AND NOT (A AND D)

2. State whether the following are valid assignment statements using boolean variables. F, I, J, K, M, T, X, Y, Z are of type integer.

   (a) Alice := FALSE;
   (b) Barbara = TRUE;
   (c) M = T > I;
   (d) Flag := F = T + 3;
   (e) Add := X < Y OR Y > Z;
   (f) Found := (X >= Y) AND (J = K + M * 4);

Topic 9

STANDARD OR BUILT-IN FUNCTIONS

Performance Objectives: Students will be able to:

- list the standard functions
- indicate the argument type and result type for each standard function
- write expressions and statements using standard functions
- write an expression to convert degree measures to radian measures and vice versa
- write an expression to round off (truncate) a real number to a given number of decimal places
- use standard functions to evaluate other commonly used mathematical functions, such as exponentiation, $\log_{10}x$, 10^x, tan (x), arcsin (x), ln (x).

Vocabulary: abs, exp, ln, sqr, sqrt, round, trunc, arctan, cos, sin, odd, pred, succ, ord, chr, eof, eoln, function

Motivation:

- You have all seen functions used in mathematics. What does the mathematical function $F(X)=X^2$ mean, and how would you evaluate F(3) for that function?

Aim: What are the standard Pascal functions and how are they used?

Development:

1. A mathematical function uses its definition and argument(s) to determine a value. The above function is defined to have a result equal to the square of its argument. Therefore, F(3) has a value of 9 (i.e., 3^2).

2. In Pascal there are standard functions which behave in the same manner. Functions have arguments which must be of a certain type, and they take on a resulting value of a specific type.

3. Below is a list of standard Pascal functions. The last two, eof and eoln, are only included for completeness. They will be discussed in a later lesson. Also, scalar types for pred and succ will be covered in a future lesson.

FUNCTION NAME	DESCRIPTION	ARGUMENT TYPE	RESULT TYPE
sqr	square	real/integer	same
sqrt	square root	real/integer	real
abs	absolute value	real/integer	same
odd	true if odd; false if even	integer	boolean
round	closest integer	real	integer
trunc	integer part of argument	real	integer
ln	log base e	real/integer	real
exp	e (2.71828) raised to a power	real/integer	real
chr	char for ordinal number	integer (ordinal number)	char
ord	ordinal number for scalar	ordinal (scalar, except real)	integer (ordinal number)
succ	value immediately following	ordinal (scalar except real)	same
pred	value immediately preceding	ordinal (scalar, except real)	same
sin	sine for angle (in radians)	real/integer(radians)	real
cos	cosine for angle (in radians)	real/integer(radians)	real
arctan	arctan	real/integer	real (radians)
eof	true if no more data in file; false otherwise	none/file name	boolean
eoln	true if end of current line; false otherwise	none/file name	boolean

Note: The UCSD compiler used on the Apple requires the insertion of the phrase "USES transcend" before the variable declaration if the functions sqrt, log, ln, arctan, exp, sin, and cos are used in the program.

4. Trignometric functions use radian measure. For degrees you must use the following constant for pi and these conversion formulas:

   ```
   CONST
     pi = 3.14159;

   degrees = radians * 180 / pi
   radians = degrees * pi / 180
   ```

5. Exponentiation can be performed using logarithms. For example:

   $(32.5)^9$ = exp (9 * ln (32.5))

   $\sqrt[3]{17}$ = exp (ln (17) / 3)

   X^Y = exp (Y * ln (X))

6. By using mathematical formulas, we can evaluate certain other commonly used mathematical functions using our standard functions. A partial list follows for reference:

   $\log_{10} X$ = ln (X) / ln (10)

   10^X = exp (X * ln (10))

   tan (X) = sin (X) / cos (X)

   sec (X) = 1 / cos(X)

   arcsin (X) = arctan (X / sqrt (1 - sqr(X)))

   arccos (X) = arctan (sqrt (1 - sqr(X)) / X)

7. The round (trunc) function can be used to round off (truncate) a real number to a given number of decimal places. For example:

   round X to 3 places = round (X * 1000) / 1000
   round X to 2 places = round (X * 100) / 100
   truncate X to 3 places = trunc (X * 1000) / 1000

8. The ord function returns the ordinal number of a character. This number will vary from computer to computer, though many computers use the ASCII code value for ordinal numbers. The chr function will return the character corresponding to the ordinal number argument. The ord, pred, and succ functions will be used in a later lesson with other types of arguments.

   One common use of ord is to change a character digit or letter to an integer value. For example, if digit is a character digit (i.e., between 0 and 9), then

```
ord (digit) - ord ('0')
```

will be the corresponding integer. Also, if letter is a character letter (i.e., between A and Z), then

```
ord (letter) - ord ('A') + 1
```

will be the corresponding integer from 1 to 26.

Applications:

1. State whether each of the following expressions is valid. If an expression is not valid, correct it.

(a) sqr (5) = 25
(b) sqrt (4) = 2.0
(c) abs (3) = 3
(d) odd (7) = true
(e) round (3.7) = 4
(f) trunc (3.9) = 3
(g) ln (1) = 0.0
(h) succ (5) = 6
(i) pred (6) = 5
(j) sqr (2.3) = 5.29
(k) sqrt (2) = 1.414
(l) abs (-2.7) = 2.7
(m) odd (8) = false
(n) round (3.4) = 3
(o) trunc (3.4) = 3
(p) exp (0) = 1.0
(q) succ ('C') = 'D'
(r) pred ('D') = 'C'

Summary:

1. What are the standard Pascal functions? Describe the type of argument and result for each of these functions.

2. How are real numbers rounded off (truncated) in Pascal?

3. What are the formulas for converting degrees to radians, and vice versa?

Exercises:

1. Write the Pascal assignment statements for:

(a) $Y = \cos (2X)$

(b) $Y = 2 \sin (X)$

(c) $Y = 3 \tan (4X)$

(d) $Y = \sin^2 (4X)$

(e) $A = |X-Y|$

(f) $B = \sqrt{7-4X}$

(g) $C = \frac{\sqrt{X+Y}}{X-Y}$

(h) $R2 = \frac{-B - \sqrt{B^2-4AC}}{2A}$

(i) $theta = \tan^{-1} (Y/X)$

(j) $X = \sin (20^\circ)$

(k) $C = \sqrt{A^2+B^2}$

2. Write the Pascal expression that can be used to round off a real number to:

   (a) 1 decimal place
   (b) 5 decimal places

3. Write a program that will input the real lengths of the two legs of a right triangle (A, B) and output the length of the hypotenuse C. (Use the pythagorean theorem: $C^2 = A^2 + B^2$.)

4. Write a program that will input the lengths of the three sides of a triangle (A, B, C) and output the area. Use Heron's formula for area which is:

   $$\text{area} = \sqrt{S(S-A)\cdot(S-B)\cdot(S-C)} \qquad \text{where } S = \frac{A+B+C}{2}$$

5. Write a program that will input the cartesian coordinates for two endpoints of a line [(X_1, Y_1) and (X_2, Y_2)] and output the measure of the line segment indicated by D, where:

   $$D = \sqrt{(X_2-X_1)^2 + (Y_2-Y_1)^2}$$

Topic 10

IF-THEN STATEMENTS

Performance Objectives: Students will be able to:

- explain how the IF-THEN statement is used by a computer to make run-time decisions
- describe how compound statements can be used in an IF-THEN statement
- diagram (flowchart) the sequence of actions in an IF-THEN statement

Vocabulary: compound statement, delimiter, IF-THEN statement, run-time

Motivation:

- A store gives a 15% discount on purchases of $100 or more. Write a program that reads the total cost of a person's purchases and prints the amount owed, including New York City sales tax of 8 1/4%. (Elicit responses.)

- What decision does the computer program have to make? (whether the total cost is less than $100, or $100 or more)

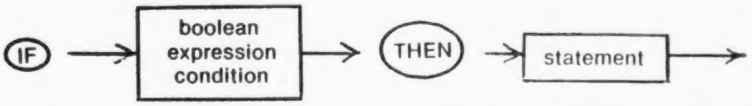
- What are the implications of this decision? (whether or not the discount will be given)

Aim: How does the computer make decisions?

Development:

1. We need the capability to execute a statement only if a condition is satisfied. The power of most meaningful programs lies in their ability to let the computer make decisions and take appropriate action during execution, run-time, based on current circumstances.

2. In Pascal, the IF-THEN statement gives us this capability. This is the syntax diagram for the IF-THEN statement:

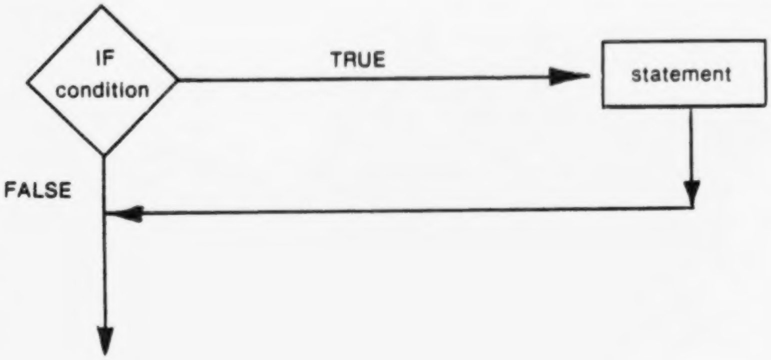


If the condition (boolean expression) is true, then the statement will be executed. If the condition is false, then the statement will not be executed.

3. How would we write an IF-THEN statement that prints "number is too large," when the variable number has a value greater than 5,000? (Elicit responses.)

```
IF number > 5000 THEN
  writeln ('Number is too large');
```

4. The actions of the IF-THEN statement can be easily understood with a flowchart-type diagram:



5. This program illustrates the use of the IF-THEN statement and is one possible solution to our original problem:

```
PROGRAM Costofpurchases (input, output);
CONST
  highcost = 100.00;
  discountrate = 0.15;
  taxrate = 0.0825;
VAR
  cost, taxes, finalcost : real;
BEGIN
  write ('Enter total price of items purchased:');
  readln (cost);
  IF cost >= highcost THEN
    cost := (1.0 - discountrate) * cost;
  taxes := taxrate * cost;
  finalcost := cost + taxes;
  writeln ('Amount owed is', finalcost : 8 : 2)
END. (* Costofpurchases *)
```

Why is it preferable to use constant declarations rather than numbers throughout the program? (The program is more readable and can more easily be modified.)

6. What would be the output for this program segment?

```
Y := 2;
N := 50;
IF N = 25 THEN
  N := N + 1;
  Y := 3;
writeln (N, Y);
```

Don't be surprised that the output is 50 3 and not 50 2. Syntactically, the THEN part of the IF-THEN statement only consists of

```
N := N + 1;
```

and is terminated by the semicolon. The next program segment is equivalent to the previous one, but the indentation more properly represents the correct meaning:

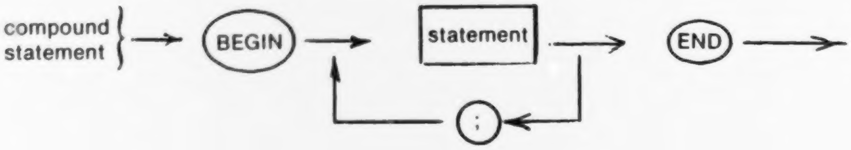
```
Y := 2;
N := 50;
IF N = 25 THEN
  N := N + 1;
Y := 3;
writeln (N, Y);
```

The statement

```
Y := 3;
```

will be executed in both cases.

7. We can include several statements together as a group (e.g., in an IF-THEN statement) by using a compound statement. A compound statement is a sequence of one or more statements starting with the word BEGIN and ending with the word END. This is a syntax diagram for a compound statement:



Since delimiters are characters or words that mark a boundary, BEGIN and END are the delimiters in a compound statement that bracket the beginning and end of the group of statements.

8. In Pascal, a compound statement can be used anywhere that a single statement can be used. Consequently, in the following program segment:

```
Y := 2;
N := 50;
IF N = 25 THEN
  BEGIN
    N := N + 1;
    Y := 3
  END;
writeln (N, Y);
```

the program will not execute

```
N := N + 1;
Y := 3
```

since the two statements are bound together in the compound statement, and the condition N = 25 is false. The output is 50 2.

Note: The semicolon is used to separate the statements in a compound statement.

9. The body of a program is actually a compound statement.

Applications:

1. Write IF-THEN statements for the following examples:

   (a) If a person's average is at least 90, write "honor student."
   (b) If a number is between 40 and 60, write "illegal number."
   (c) IF X equals 2 or Y equals 3, increment Z by 1.
   (d) If num is less than 10, then double num and write "The number is doubled."
   (e) If the boolean variable valid is true, then set bottom equal to 0 and top equal to 100.

   Answers:

```
(a)  IF average>= 90 THEN
       writeln ('honor student');

(b)  IF (number > 40) AND (number< 60) THEN
       writeln ('illegal number');

(c)  IF (x = 2) OR (y = 3) THEN
       Z := Z + 1;

(d)  IF num < 10 THEN
       BEGIN
         num := 2 * num;
         writeln ('The number is doubled')
       END;

(e)  IF valid THEN
       BEGIN
         bottom := 0;
         top := 100
       END;
```

Note: A misplaced semicolon after the word THEN often causes problems that are hard to diagnose. Another common error is to neglect to express the THEN statement as a compound statement.

Summary:

1. How does the computer use IF-THEN statments to make decisions?
2. What is a compound statement?
3. What are the delimiters for a compound statement?
4. Where in a Pascal program can a compound statement be used?

Exercises:

1. Write a program that reads two real numbers and prints out the larger number. (Use only one IF-THEN statement.)
2. Write a program that reads the coefficients of a quadratic equation and outputs the real roots (if any exist).
3. Write a program that reads four test grades and outputs the average. If the average is above 90, also print the messages "Excellent!" and "Keep up the good work!" on separate lines.
4. Write a program to determine the amount of change due when a bill is paid. You should input the amount of the bill and the amount paid. The program should compute and print the change due and its breakdown into dollar bills, quarters, dimes, and pennies.

Topic 11
IF-THEN-ELSE CONTROL STRUCTURES

Performance Objectives: Students will be able to:

- explain and diagram the actions of an IF-THEN-ELSE statement
- use the IF-THEN-ELSE statement in programs that involve decisions between two alternative courses of action
- show how a value can be assigned to a boolean variable with an arithmetic assignment statement and then use this boolean variable in an IF-THEN-ELSE statement to make the coding read very much like English

Vocabulary: IF-THEN-ELSE statement, control structure, boolean variable

Motivation:

- Write a Pascal program that will read in two integers and print the numbers in order with the smaller number first. (Elicit responses.)

- This is one solution to the problem:

```
PROGRAM Swap (input, output);
VAR
  num1, num2 : integer;
BEGIN
  IF num1 < num2 THEN
    writeln (num1, num2);
  IF num1 >= num2 THEN
    writeln (num2, num1)
END.  (* Swap *)
```

- What decisions must be made by the computer program? (whether num1 is greater than num2 and whether num1 is less than num2)

- One of the two writeln statements will be executed, depending on the truth value of the two given conditions. Since there are two possible choices, Pascal allows for two alternative outcomes.

Aim: How does the computer make decisions in which two alternative outcomes can be executed?

Development:

1. What structure permits the computer to decide between two alternatives? (Elicit responses.)

```
IF condition THEN                    IF condition
  statement 1              or          THEN statement 1
ELSE                                   ELSE statement 2;
  statement 2;
```

These illustrate two different formats.

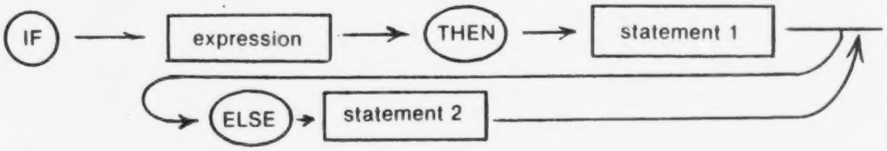
2. When will statement 1 be executed?

   If the condition is true, the statement following THEN (statement 1) will be executed.

3. When will statement 2 be executed?

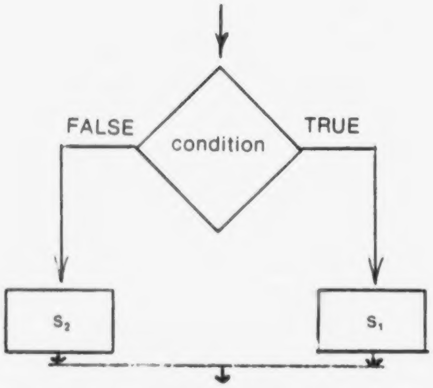
   If the condition is false, the statement following ELSE (statement 2) will be executed.

4. This is the syntax diagram for the IF-THEN-ELSE statement:



In an IF-THEN statement, the THEN is mandatory but the ELSE part is optional.

5. The action of the IF-THEN-ELSE statement can also be understood by looking at a flowchart diagram:



IF the condition is true, the statement following the THEN will be executed (S_1), ELSE, IF the condition is false, the statement following the ELSE will be executed (S_2).

Note: Be sure not to put semicolons before or after THEN, statement 1, or ELSE. The action of the IF-THEN-ELSE statement concludes at the semicolon that follows statement 2.

6. How would you rewrite program Swap using an IF-THEN-ELSE control structure? (Elicit responses.)

This is one solution:

```
PROGRAM Swap (input, output);
VAR
  num1, num2 : integer;
BEGIN
  read (num1, num2);
  IF num1 < num2 THEN
    writeln (num1, num2)
  ELSE
    writeln (num2, num1)
END. (* Swap *)
```

Note: There is no semicolon in this IF-THEN-ELSE statement because the END of the main program terminates the action of the control structure.

7. Suppose you want to compute and print an employee's gross pay. Include overtime pay, which is time and a half if hours are greater than 40. If there is no overtime, have the program compute the regular gross pay. (Elicit responses.)

This is one solution:

```
IF hours > 40 THEN
  gross := 40 * rate + (hours - 40) * (rate * 1.5)
ELSE
  gross := rate * hours;
```

It is, however, usually easier to understand a calculation if it is done in several steps. For instance, the computation for gross pay includes computing overtime hours and overtime rate. Let's see how the original solution can be broken down into these component calculations:

. How many hours overtime did the individual work?

```
overtime := hours - 40;
```

. What is the new rate for the overtime hours?

```
overrate := rate * 1.5;
```

. What is the overtime pay?

```
overpay := overtime * overrate;
```

. How much will an individual be paid for the first 40 hours?

```
regpay := 40 * rate;
```

. How can we calculate the entire gross pay for an individual who worked more than 40 hours?

```
gross := overpay + regpay;
```

8. As in the IF-THEN statement, we can combine several statements together into a compound statement using the reserved words BEGIN and END to bracket the executable statements that make up the compound statement:

```
IF condition THEN
  BEGIN
    statement 1;
    statement 2
  END
ELSE
  BEGIN
    statement 1;
    statement 2
  END;
```

9. How can we use this convention to rewrite the program segment for overtime pay? (Elicit responses.)

```
IF Hours > 40 THEN
  BEGIN (* compute salary for overtime worker *)
    overtime := hours - 40.0; (* overtime hours *)
    overrate := rate * 1.5; (* overtime rate *)
    overpay := overtime * overrate, (* overtime pay *)
    regpay := rate * 40.0; (* pay for first 40 hours *)
    gross := overpay + regpay (* salary including overtime *)
  END
ELSE (* compute salary for worker with no overtime *)
  gross := rate * hours;
```

10. In an IF structure, the condition is always evaluated as true or false. For example:

```
IF discriminant >= 0 THEN (* find the roots of the equations *)
```

- How can this condition be assigned to a boolean variable in an assignment statement?

```
rootexists := (discriminant >= 0)
```

- How can this boolean variable be incorporated into an IF structure?

```
IF rootexists THEN (* find the roots of the equations *)
```

Application:

1. Print a message that says "You have a passing average" if a student's average is 65 or over, and print a message that says "You have a failing average" if the student's average is less than 65.

Possible solution:

```
IF average >= 65 THEN
  writeln ('You have a passing average')
ELSE
  writeln ('You have a failing average');
```

2. A senior citizen is a person whose age is 65 or older. A certain store allows senior citizens to receive a 10% discount on any item over $10. Write a program that determines if a discount is allowed and prints an appropriate message with the cost of the item.

Possible solution:

```
PROGRAM Seniordiscount (input, output);
CONST
  discrate = 0.10; (* discount rate *)
  minprice = 10.00; (* minimum eligibility price for discount *)
VAR
  itemprice, (* price of item *) cost : real;
  (* final cost of item *)
  response : char; (* Y or N to question *)
  seniorcitizen, (* assignment variable for response *)
  priceiseligible : boolean; (* assignment variable for
  itemprice > $10 *)
BEGIN
  writeln ('Are you a senior citizen?  answer Y for yes
  or N for no');
  readln (response);
  seniorcitizen := response = 'Y';
  writeln ('What is the price of your item?');
  readln (itemprice);
  priceiseligible := itemprice > minprice;
  IF seniorcitizen AND priceiseligible THEN
    (* calculate cost of item including discount *)
    BEGIN
      writeln ('You qualify for a discount');
      cost := itemprice - itemprice * discrate
    END
  ELSE (* no discount *)
    BEGIN
      writeln ('You do not qualify for a discount');
      cost := itemprice
    END;
  writeln ('The cost of the item is', cost : 7 : 2)
END.  (* Seniordiscount *)
```

Summary:

1. What kind of decisions would call for the use of an IF-THEN-ELSE statement?

2. In an IF-THEN-ELSE statement, when will the statement following THEN be executed? When will the statement following ELSE be executed?

3. How can we assign a value to a boolean variable? How can we use that boolean variable in an IF-THEN-ELSE statement to make our coding read like English?

4. What are the two major bugs to be aware of in writing an IF-THEN-ELSE statement?

__Exercise:__

1. Write a program to read in any real number and write the square root of the real number in simplest form. (Imaginary numbers should be expressed in terms of i.)

Topic 12
NESTED IF-THEN-ELSE STATEMENTS

Performance Objectives: Students will be able to:

- define a compound IF statement (nested conditional statement)
- diagram compound IF statements from program code and write program code from diagrams
- state the rule by which the computer deals with the apparent ambiguity of the dangling ELSE clause

Vocabulary: nested IF-THEN-ELSE statement, compound IF statement

Motivation:

- What will this program do? (Elicit responses.)

```
PROGRAM Grades (input, output);
VAR
  average : real; (* computed average *)
BEGIN (* Grades *)
  readln (average);
  IF average >= 85 THEN
    writeln ('Excellent Grade')
  ELSE
    IF average >= 65 THEN
      writeln ('Passing Grade')
    ELSE
      writeln ('Failing Grade')
END. (* Grades *)
```

- This program will print out different messages, depending on the average grade. How many distinct messages can be written? (3) How many choices must be made by the computer program? (3)

- Since computer programs like this one have the ability to make more than two decisions, Pascal allows for multiway alternatives with the nested IF-THEN-ELSE statement, also referred to as the compound IF statement.

Aim: How can the IF-THEN-ELSE statement be compounded to allow the computer to make more complex decisions?

Development:

1. A nested IF-THEN-ELSE statement (compound IF statement) occurs when the statement following the reserved word THEN or ELSE is another IF statement.

2. The following is the general form of the nested IF-THEN-ELSE statement:

ELSE clause of IF statement is an IF statement:

```
IF condition 1 THEN
  statement 1
ELSE
  IF condition 2 THEN
    statement 2
  ELSE
    statement 3
```

3. A chart can be used to see the execution of each statement:

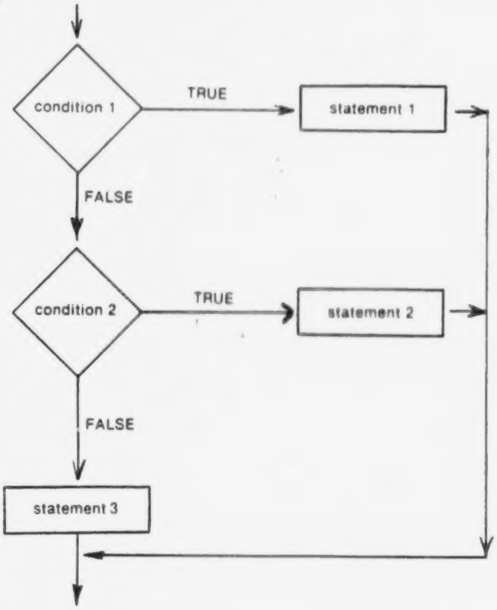
condition 1	condition 2	statement 1	statement 2	statement 3
T	-	✓	-	-
F	T	-	✓	-
F	F	-	-	✓

- IF condition 1 is true, THEN statement 1 will be executed and control will direct itself to the next sequential statement following the compound IF statement.

- IF condition 1 is false and condition 2 is true, THEN statement 2 will be executed and likewise control will direct itself to the next sequential statement.

- IF condition 1 through condition n-1 are false (i.e., none of the conditions is true), THEN statement n will be executed, and control again will direct itself to the next sequential statement.

4. The action of the compound IF statement can be understood with a flowchart diagram:



Notice that if all n-1 conditions are false, statement n will be executed.

5. We can now understand what program Grades will do. It will write "Excellent Grade" if the average >= 85. If that condition is false, it will check if the average >= 65. If the average >= 65 (i.e., the condition is true), then "Passing Grade" will be written; if not, "Failing Grade" will be written.

6. A compound IF statement also occurs when the THEN clause of the IF statement is itself an IF statement. This is the general form for this kind of compound IF statement:

```
IF condition 1 THEN
  IF condition 2 THEN
    IF condition 3 THEN
      Statement 1
    ELSE
      Statement 2
  ELSE
    statement 3
ELSE
  statement 4
```

. When will statement 1 be executed?

- When condition 1, condition 2, and condition 3 are true.

. When will statement 2 be executed?

- When condition 1 and condition 2 are true, and condition 3 is false.

. When will statement 3 be executed?

- When condition 1 is true and condition 2 is false.

. When will statement 4 be executed?

- When condition 1 is false.

7. How can the following nested IF be rewritten in a more compact and readable fashion? (Elicit responses.)

```
IF condition 1 THEN
  IF condition 2 THEN
    IF condition 3 THEN
      statement 1;
```

Since statement 1 will only be executed when all three conditions are true, it is usually written as:

```
IF condition 1 AND condition 2 AND condition 3 THEN
  Statement 1
```

8. When a programmer writes a program, how does the indentation affect the program when it is compiled?

   It doesn't. Proper indentation is used to improve the readability of the program for the programmer. The compiler does not understand or care about indentation.

9. How does the compiler know which IF-THEN-ELSE statements are paired together?

   An ELSE belongs to the nearest preceding IF that is not already paired with an ELSE.

10. Using this convention, how would you write the program segment for this IF-THEN-ELSE statement? (Elicit responses.)

```
IF condition 1 THEN IF condition 2 THEN statement 3 ELSE statement 4
                    └──────────────────────────────────┘
```

   Answer:

```
IF condition 1 THEN
  IF condition 2 THEN
    statement 3
  ELSE
    statement 4
```

   Start at the right. Look for the last ELSE. The ELSE should be paired with the nearest preceding IF that isn't already paired with an ELSE.

11. Since the compiler does not pay attention to indentation, how can we resolve the problem of the programmer's "dangling" ELSE, as in the following example:

```
IF cond 1 THEN
  IF cond 2 THEN
    $S_1$
ELSE (* dangling else *)
    $S_2$
```

   Which IF does the ELSE belong to? From the indentation, it appears that S_2 will be executed when cond 1 is false. However, using the rule for pairing ELSE and IF shows that ELSE S_2 is really coupled with the second IF, and that this indentation better represents the IF-THEN-ELSE relationship:

```
IF cond 1 THEN
  IF cond 2 THEN
    $S_1$
  ELSE
    $S_2$
```

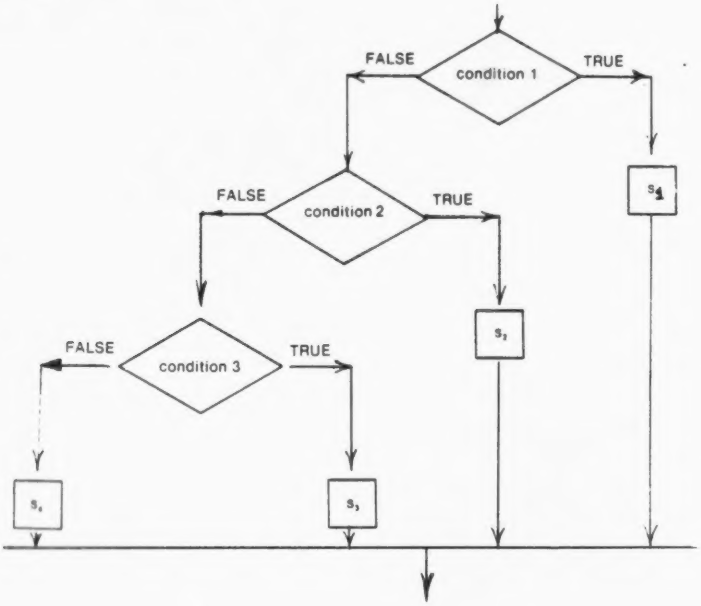
Applications:

1. Rewrite the following IF-THEN-ELSE statement with proper indentation. Make a flowchart diagram.

IF cond 1 THEN S_1 ELSE IF cond 2 THEN S_2 ELSE IF Cond 3 THEN S_3 ELSE S_4

Answer:

```
IF cond 1 THEN
  $S_1$
ELSE
  IF cond 2 THEN
    $S_2$
  ELSE
    IF cond 3 THEN
      $S_3$
    ELSE
      $S_4$
```

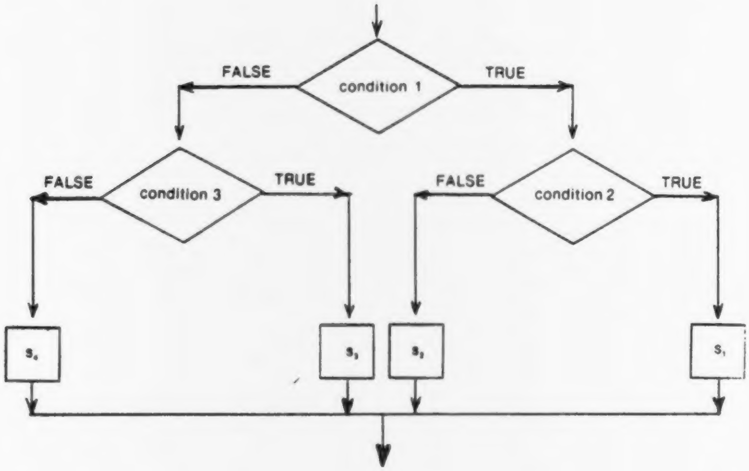

FALSE
condition 1
TRUE
FALSE
condition 2
TRUE
S_1
FALSE
condition 3
TRUE
S_2
S_4
S_3

2. Rewrite the following IF-THEN-ELSE statement with proper indentation. Make a flowchart diagram.

IF cond 1 THEN IF cond 2 THEN S_1 ELSE S_2 ELSE IF Cond 3 THEN S_3 ELSE S_4

Answer:

```
IF cond 1 THEN
  IF cond 2 THEN
    $S_1$
  ELSE
    $S_2$
ELSE
  IF cond 3 THEN
    $S_3$
  ELSE
    $S_4$
```


condition 1
FALSE
TRUE
FALSE
condition 3
TRUE
FALSE
condition 2
TRUE
S4
S3
S2
S1

Summary:

1. What is a compound IF statement?
2. What is the rule for pairing an ELSE with an IF?
3. How does the compiler deal with the ambiguity of the dangling ELSE?

Topic 13

PROGRAMS INVOLVING IF-THEN-ELSE STATEMENTS

Performance Objectives: Students will be able to:

- use IF-THEN-ELSE statements in computer programs
- identify problems that arise through improper use of compound statements

Vocabulary: nested statement, slope, parallel lines

Motivation:

- Write a program that reads the coordinates of four consecutive vertices of a quadrilateral. Your program should determine whether the quadrilateral is a parallelogram, rhombus, rectangle, square, trapezoid, or none of these. Do not deal with the special quadrilateral that has at least one side parallel to the Y axis. (Elicit responses.)

- What geometric properties must the computer program check to determine the type of figure?
    - For the figure to be a trapezoid, one and only one pair of opposite sides must be parallel.
    - To be a parallelogram, both pairs of opposite sides must be parallel.
    - To be a rectangle, the figure must be a parallelogram and contain one right angle.
    - To be a rhombus, the quadrilateral must be a parallelogram with two equal adjacent sides.
    - To be a square, the quadrilateral must be a rectangle with two equal adjacent sides.

- How can we write the computer program so that it will determine if these properties hold?
    - Use nested IF-THEN-ELSE statements.

Aim: How do we write programs that use IF-THEN-ELSE statements?

Development:

1. We will first need the rules for the computer to use in determining the shape of the geometric figure:
    - How do you test for parallel sides?
        - Use the slope formula:

          $$M = (Y2 - Y1) / (X2 - X1).$$

        - Remember, process only those slopes where the denominator (X2 - X1) does not equal zero (i.e., no side is parallel to the Y axis).

- How do you test for a right angle?

  - The product of the slopes of two adjacent sides must be negative reciprocals of each other.

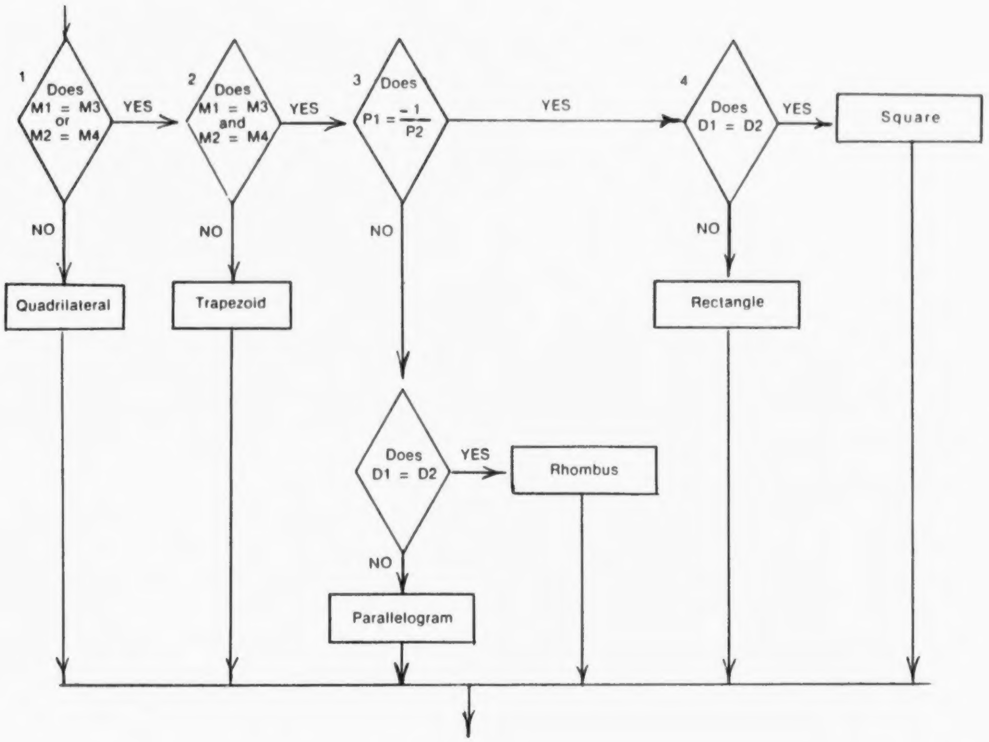
- How do you test for equal sides:

  - Use the distance formula: d = sqrt (sqr (X2 - X1) + sqr (Y2 - Y1))

2. What is the first decision that the computer program should make, and why?

   - The program should decide if the difference between any set of consecutive X coordinates is zero. Since it is necessary to find the slopes of all four sides of the quadrilateral, the computer will give a run-time error if the denominator is zero. To avoid the run-time error, have the program print a message that quadrilaterals where one side is parallel to the Y axis will not be considered in this example.

3. After the computer program decides that none of the sides are parallel to the Y axis, it must then make several decisions, in proper sequence, to determine the type of quadrilateral. How can these decisions be depicted with a flowchart? (Elicit responses.)


1
Does M1 = M3 or M2 = M4
YES
NO
Quadrilateral
2
Does M1 = M3 and M2 = M4
YES
NO
Trapezoid
3
Does P1 = $\frac{-1}{P2}$
YES
NO
4
Does D1 = D2
YES
NO
Square
Rectangle
Does D1 = D2
YES
NO
Rhombus
Parallelogram

- Is there one pair of opposite sides that are parallel? (Are the slopes of one pair of opposite sides equal?) If yes, check condition 2. If no, then the figure is a quadrilateral.

- Are both pairs of opposite sides parallel? (Are the slopes of each pair of opposite sides equal?) If yes, check condition 3. If no, then the figure is a trapezoid.

- Is there a right angle? (Are the slopes negative reciprocals of each other)? If yes, check condition 4.

- Are two adjacent sides equal in length? (Use the distance formula.) If yes, then the figure is a square. If no, then the figure is a rectangle. If no, check condition 5.

- Are two adjacent sides equal in length? (Use the distance formula.) If yes, then the figure is a rhombus. If no, then the figure is a parallelogram.

4. In each case, after the type of figure is established, control will pass to the next statement in the program.

What are the necessary calculations for each condition?

- Condition 1: Is (M1 = M3) or (M2 = M4)?

```
M1 := (Y2 - Y1) / (X2 - X1)
M2 := (Y3 - Y2) / (X3 - X2)
M3 := (Y4 - Y3) / (X4 - X3)
M4 := (Y1 - Y4) / (X1 - X4)
```

- Condition 2: Is (M1 = M3) and (M2 = M4)?

  - Use the same calculation as above.

- Condition 3: Is there a right angle?

  - To check for a right angle, see if the slopes of two adjacent sides are negative reciprocals of each other. (Does M1 = -1/M2? That is, does the product of the slopes = -1, or does M1 * M2 = -1?)

- If M1 is calculated to be 1/2 and M2 is calculated to be -2/1, the computer would convert it into decimal form:

```
M1 ←—— 0.50000
M2 ←—— -2.00000
```

These numbers would not cause a problem since

```
M1 * M2 = -1

0.50000 * 2.00000 = -1.0000
```

5. Suppose the slopes were

M1 = 3/4
M2 = -4/3

Mathematically, 3/4 * -4/3 = -1.

. What problems arise with the computer?

- Since M1 = 3/4, the computer will convert that to M1 = 0.75000, and likewise M2 = -4/3 will be converted to M2 = -1.33333. By multiplying M1 by M2 (-1.33333 * 0.75000), the computer does not get an exact answer. It will get an approximation, very close to -1, but not equal to -1. What is the reason? Since -4/3 is a repeating decimal, the computer truncates the decimal portion, thus making the final answer an approximate and not an exact answer. (The computer actually deals with binary numbers. The decimal .2, for example, is an infinite binary decimal equal to $.\overline{0011}$.

. What can be done to alleviate this problem?

- Slope is defined as

$$\text{slope}_1 = \frac{\Delta Y}{\Delta X} = \frac{Y_2 - Y_1}{X_2 - X_1}$$

$$\text{slope}_2 = \frac{\Delta Y}{\Delta X} = \frac{Y_3 - Y_2}{X_3 - X_2}$$

- Using the same idea that the slopes must be negative reciprocals of each other, then

$$\frac{(Y_2 - Y_1)}{(X_2 - X_1)} = - \frac{(X_3 - X_2)}{(Y_3 - Y_2)}$$

must hold in order for there to be a right angle. The above expression is equivalent to

$$(Y_2 - Y_1) * (Y_3 - Y_2) = - (X_3 - X_2) * (X_2 - X_1)$$

- Going back to the original set of numbers that gave the problem, where

$$\frac{\Delta Y}{\Delta X} = \frac{Y_2 - Y_1}{X_2 - X_1} = \frac{3}{4} \quad \text{and} \quad \frac{\Delta Y}{\Delta X} = \frac{Y_3 - Y_2}{X_3 - X_2} = \frac{-4}{3}$$

$$\frac{(Y_2 - Y_1)}{(X_2 - X_1)} = - \frac{(X_3 - X_2)}{(Y_3 - Y_2)}$$

$$(Y_2 - Y_1) * (Y_3 - Y_2)= - (X3 - X2) * (X2 - X1)$$

$$3 * -4 = -3 * 4$$

$$-12 = -12$$

- By using the equivalent expression

$$(Y_2 - Y_1) * (Y_3 - Y_2) = -(X_3 - X_2) * (X_2 - X_1)$$

as the condition for a right angle, the problem of a repeating decimal will be solved.

<u>Note:</u> See Topic 7 for an alternative approach.

6. How can we now code the program?

. What variables must be declared?

- the X and Y coordinates of the quadrilateral, the slopes of the 4 sides, the lengths of the two adjacent sides

. What must be done prior to the computer program testing all the conditions?

- Check that none of the sides are parallel to the Y axis.
- Read the coordinates of the vertices.
- Calculate the slopes and distances (length).

. Begin testing (nested IF-THEN-ELSE) for the different types of figures:

```
PROGRAM Testquad (input, output);
(* This program will determine whether the given vertices of *)
(* a quadrilateral form a trapezoid, parallellogram, rectangle *)
(* rhombus, or square.  The special quadrilateral where at least *)
(* one side is parallel to the Y axis will not be considered. *)
VAR
  (* X and Y coordinates of the quadrilateral *)
  X1, Y1, X2, Y2, X3, Y3, X4, Y4 : real;
  (* slopes of the sides *)
  M1, M2, M3, M4 : real;
  (* length of 2 adjacent sides *)
  D1, D2 : real;
BEGIN (* Testquad *)
  write('Please enter the X and Y coordinates of four');
  writeln(' consecutive vertices of a quadrilateral');
  readln(X1, Y1, X2, Y2, X3, Y3, X4, Y4);
  (* determine if any of the sides are parallel to the Y axis *)
  IF (X1 - X2 <> 0) AND (X2 - X3 <> 0) AND (X3 - X4 <> 0) AND
  (X4 - X1 <> 0) THEN
```

```
        BEGIN
          (* calculate the slope of each side *)
          M1 := (Y2 - Y1) / (X2 - X1);
          M2 := (Y3 - Y2) / (X3 - X2);
          M3 := (Y4 - Y3) / (X4 - X3);
          M4 := (Y1 - Y4) / (X1 - X4);
          (* calculate the length of two adjacent sides *)
          D1 := sqrt (sqr (X2 - X1) + sqr (Y2 - Y1));
          D2 := sqrt (sqr (X3 - X2) + sqr (Y3 - Y2));
          (* test for trapezoid, one pair of parallel sides *)
          IF(M1 = M3) OR (M2 = M4) THEN
            (* test for parallelogram, both sides parallel *)
            IF(M1 = M3) AND (M2 = M4)THEN
              (* test for rectangle, one right angle *)
              IF (Y2 - Y1) * (Y3 - Y2) = -(X3 - X2) * (X2 - X1) THEN
                (* test for square, 2 adjacent sides equal *)
                IF D1 = D2 THEN
                  writeln('The figure is a square.')
                ELSE (* the 2 adjacent sides are not equal *)
                  writeln('The figure is a rectangle.')
              ELSE (* there is no right angle *)
                (* test for rhombus, 2 adjacent sides equal *)
                IF D1 = D2 THEN
                  writeln('The figure is a rhombus.')
                ELSE (* the two adjacent sides are not equal *)
                  writeln('The figure is a parallelogram.')
            ELSE(* both pairs of opposite sides are not parallel *)
              writeln('The figure is a trapezoid')
          ELSE(* there are no parallel sides *)
            writeln('The figure is a quadrilateral')
        END
      ELSE
        BEGIN
          write('Quadrilaterals where at least one side is');
          writeln('parallel to the Y axis will not be considered')
        END
    END. (* Testquad *)
```

Summary:

1. What are the properties that determine whether a quadrilateral is a trapezoid, parallelogram, rectangle, or square?

2. What mathematical relationship must exist in Pascal for an angle to be a right angle? Why can't the following relationship be used in Pascal statements to determine if an angle is right?

   slope 1 x slope 2 = -1

Topic 14
REPEAT-UNTIL STRUCTURES

Performance Objectives: Students will be able to:

- use the REPEAT-UNTIL statement
- define a delimiter and give examples in Pascal

Vocabulary: delimiter, REPEAT-UNTIL, control statement, boolean expression

Motivation:

- What does this program do?

```
PROGRAM Looper (input, output);
VAR
  number : real;
BEGIN
  writeln;
  REPEAT
    writeln ('Enter a number');
    readln (number);
    writeln (number : 8, sqr (number) : 10)
  UNTIL number < 0
END. (* Looper *)
```

- Program Looper will get a number from input and print it along with its square. The program will repeat until the user enters a negative number. This is known as a loop.

Aim: How can we use the REPEAT-UNTIL statment to control the number of times a program segment is repeated?

Development:

1. The loop in program Looper incorporates a REPEAT-UNTIL structure. Note that the body of the loop consists of multiple statements and yet is not bracketed between the delimiters BEGIN and END. Delimiters mark the boundary of a compound statement. What delimiters do you suppose replace the usual BEGIN-END delimiters here?

   - REPEAT and UNTIL serve this function.

2. The body of the loop will be executed even if the user enters a negative number. The condition for exiting is not tested until after the statements in the loop are executed. Then, if the condition following the UNTIL is true--and only then--will the loop will be terminated. As long as the condition is false, the body of the loop will be repeated. In general:

```
REPEAT
  statement 1;                    body
  statement 2;                    of
  ...                             loop
  statement n;
UNTIL boolean condition;
```

3. Can the program be modified so that the square of the negative input will not be printed?

   - Yes, read at the end of the loop just before UNTIL.

4. How will the first data entered be processed?

   - by reading the first positive value before the REPEAT-UNTIL structure

5. Rewrite part of the program so that when a negative value is entered, it will not be processed. (Elicit responses.)

   . We can handle the problem this way:

```
write ('Enter a positive value ');
readln (number);
REPEAT
  writeln (number : 8, sqr (number) : 10);
  writeln;
  write ('Enter a number ');
  readln (number)
UNTIL number < 0;
```

6. Why is there no semicolon following the last readln?

   - Since UNTIL, like END, serves as a delimiter, the semicolon is not needed to separate the readln from the following statements. UNTIL serves that function. The UNTIL also controls which statement will next be executed.

7. Find the error in the following:

```
X := 1;
REPEAT
  X := X + 2;
  writeln (X)
UNTIL X = 100;
```

   - This is an infinite loop because X will always be odd. When using a REPEAT-UNTIL structure, always make sure the condition will ultimately be satisfied.

8. Why do we write UNTIL X = 0 and not UNTIL X := 0?

   - because the expression that follows the UNTIL is a boolean expression and not an assignment statement

9. What will this segment do?

```
REPEAT
  read (char)
UNTIL char <> ' ';
```

   - It will read a character from input but ignore the keyboard entry of the space bar.

Applications:

1. Write a program that asks the user for an even integer. If the integer entered is odd, keep requesting an even integer until such an integer appears.

   Possible solution:

```
PROGRAM Even (input, output);
VAR
  evenint : integer;
BEGIN
  REPEAT
    writeln ('Enter an even integer');
    readln (evenint)
  UNTIL NOT odd (evenint);
  writeln ('Thank you')
END. (* Even *)
```

Summary:

1. The REPEAT-UNTIL structure can be used to create a loop that will repeat until the boolean expression following the UNTIL is true.

2. A compound statement need not be bracketed by the delimiters BEGIN and END, because the reserved words REPEAT and UNTIL serve as delimiters.

3. The body of a REPEAT structure is always executed at least once. Be sure the UNTIL control condition will at some point be TRUE to avoid an infinite loop.

Exercises:

1. Write a program to print the values from 10 down to 0 followed by the words blast off.

2. Write a program that will give instructions (make up your own), and then ask the operator if he or she would like to see the instructions again. Keep repeating the instructions until the operator types N.

3. Write a program to count the number of digits in a given integer. (Hint: Keep dividing the number by 10 until N DIV 10 yields a zero.)

Topic 15
WHILE-DO STATEMENTS

Performance Objectives: Students will be able to:

- explain the syntax and operation of the WHILE-DO statement
- use the WHILE-DO statement in computer programs
- state the purpose of a sentinel value
- state the difference between the WHILE-DO and REPEAT-UNTIL statements

Vocabulary: WHILE-DO, sentinel value, trailer

Motivation:

- What is wrong with the following program? (Elicit responses.)

```
PROGRAM Checktoolate (input, output);
VAR
  number : real;
BEGIN
  writeln ('Enter a positive number');
  readln (number);
  REPEAT
    writeln ('The square root of  ', number : 10 : 2, ' is ',
    sqrt (number) : 10 : 2 );
    writeln;
    writeln ('Enter a positive number');
    readln (number)
  UNTIL number < 0;
  writeln ('Sorry, but I'm not into square roots of negative numbers')
END. (* Checktoolate *)
```

- The problem with the program is that if the first number entered is negative, the body of the loop will still be executed. This is because the test for a negative input does not come until the end of the REPEAT structure.

- Pascal has a control structure that tests a condition at the beginning of a loop.

Aim: How do we use the WHILE-DO statement to test the entry/exit condition prior to executing the body of the loop?

Development:

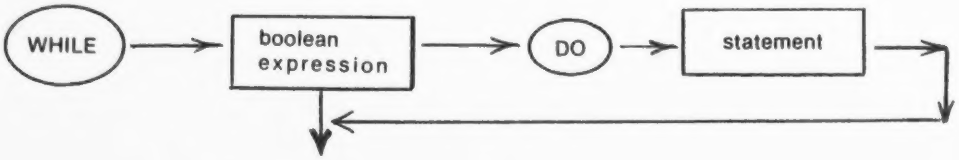
1. In program Checktoolate, replace REPEAT with WHILE number > 0 DO, eliminate the UNTIL statement, and enclose the body of the loop with a BEGIN and END. What effect will this have? (Elicit responses.)

2. This is the general form of the WHILE-DO structure:

```
WHILE boolean expression DO
  statement;
```

The body of the loop will not be executed even once unless the boolean expression is true.

3. This is the syntax diagram for the WHILE-DO structure:



4. Let's now write a program that will find the average of several numbers. We will write a batch-oriented program in which the end of each line of data will be marked by a negative value. Any such value that notes the end of a line of data is called a sentinel, or trailer. We will use a sentinel of -1 to mark the end of the numbers we want to average.

```
PROGRAM Average (input, output);
CONST
  sentinel = -1;
VAR
  mean, number, sum : real;
  count : integer;
BEGIN
  count := 0;
  sum := 0.0;
  read (number);
  WHILE Number <> sentinel DO
    BEGIN
      sum := sum + number;
      count := count + 1;
      read (number)
    END; (* WHILE *)
  mean := sum / count;
  writeln ('The average =', mean : 10 : 2)
END. (* Average *)
```

5. The WHILE-DO structure can also be used as a time delay by asking the computer to count to a given value; for example:

```
X := 0;
WHILE X < 1000 DO
  X := X + 1;
```

In this example, the body of the loop is a single statement, which obviates the use of BEGIN-END delimiters. The end of a WHILE-DO structure that consists of a single statement is marked by a semicolon. Be sure to initialize the testing variable before entering the loop (i.e., X := 0).

6. What are the differences between WHILE-DO and REPEAT-UNTIL loops? Let's compare these two program segments:

```
X := 11;
WHILE X <= 10 DO
  BEGIN
    writeln (X);
    X := X + 1
  END;


X := 11;
REPEAT
  writeln (X);
  X := X + 1
UNTIL X > 10;
```

Since X is already 11, the WHILE loop will not be executed. On the other hand, the REPEAT loop will be executed once before it is noted in the UNTIL statement that X has already exceeded 10.

7. When should we use a REPEAT loop instead of a WHILE loop? If the exit condition is dependent on the values that are first derived in the body of the loop, the REPEAT structure is preferable. On the other hand, if we don't want to enter a loop unless certain preconditions are met, the WHILE-DO structure should be the programmer's choice.

Summary:

1. A sentinel value is a way of checking for the end of an input list.

2. The WHILE-DO structure, unlike the REPEAT-UNTIL structure, tests the boolean condition at the beginning of the loop rather than the end.

3. If the body of the WHILE-DO is a compound statement, it must be bracketed by BEGIN-END delimiters. If it is a single statement, then only a semicolon is needed to mark the end of the WHILE-DO structure.

Exercises:

1. Write a program that prints 70 *'s on a line.
2. Write a program that sums the counting integers up to the point where the sum first exceeds 1,000. Determine the final counting integer that first brings the total over 1,000.
3. Write a program that prints the powers of 2 up to 10,000.
4. Write a batch-oriented program (one that reads data cards) that will read and print numbers as long as the numbers are in increasing order.
5. Write a batch-oriented program that will read a number and print it along with its square. Assume the data card ends with a sentinel value of -999.

Topic 16

FOR-DO STATEMENTS

Performance Objectives: Students will be able to:

- explain the syntax and operation of the FOR-DO statement
- use the FOR-DO statement in computer programs
- identify the difference between the FOR-DO statement and other control statements
- use nested loops in writing programs

Vocabulary: counter variable, ordinal, FOR-DO statement, DOWNTO, nested loop

Motivation:

- Use a REPEAT-UNTIL loop to print 30 *'s on a line. Then use a WHILE-DO loop to accomplish the same thing. (Elicit responses.)

```
X := 0;                      X := 0;
REPEAT                       WHILE X < 31 DO
  X := X + 1;                  BEGIN
  write ('*')                    X := X + 1;
UNTIL X := 30;                   write ('*')
                               END;
```

- Pascal has a third looping structure that is similar to the FOR-NEXT structure of BASIC.

Aim: How is the FOR-DO statement used in Pascal programming?

Development:

1. How would the program segment above be written with a FOR-DO loop?

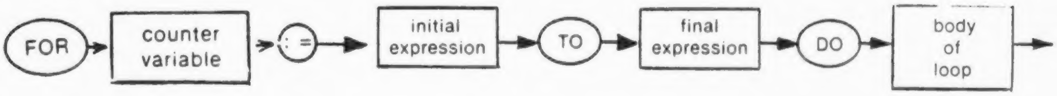
```
FOR X := 1 TO 30 DO
  write ('*');
```

The most obvious advantage here is the simplicity of the loop. The initialization (X := 1) and exit testing (X := 30) are built into one line:

```
FOR X := 1 TO 30 DO
```

2. Why does Pascal include the other looping structures? The FOR-DO loop is only practical when it is known prior to entering the loop exactly how many repetitions are needed and each succeeding value of the counter variable is the successor (succ) of the preceding value.

3. This is the structure of the FOR-DO loop:



The initial and final values may be constants, variables, or expressions of type integer. They may also be of type boolean or character, but reals cannot be used because they are not ordinal and cannot be denumerated. Only values for which there exists a predecessor or a successor may be used.

4. In the REPEAT and WHILE structures, the control variable must be modified within the loop, so that the loop will eventually terminate. However, the counter variable in a FOR-DO structure <u>must not</u> be modified in the body of the loop. If the loop is initially set for 30 repetitions, it will be processed all 30 times. If the initial value is already greater than the final value, the loop will not be processed at all.

5. Pascal has a modification of the FOR-DO structure that allows the loop variable to decrement during the repetition. To count down from 10 to 1:

```
FOR X := 10 DOWNTO 1 DO
  writeln ('X');
```

Note that:

- <u>DOWNTO</u> is one word and a reserved word.
- The FOR statement, unlike the IF statement, uses :=, since an assignment is being made.
- The value of the counter variable is not defined at the exit from the loop (in standard Pascal).
- Counter variables must be declared.
- <u>TO</u> is a reserved word.
- If the loop control variable is ascending in value, the final value must be greater than or equal to the initial value.
- If the loop control variable is descending in value, the final value must be less than or equal to the initial value.

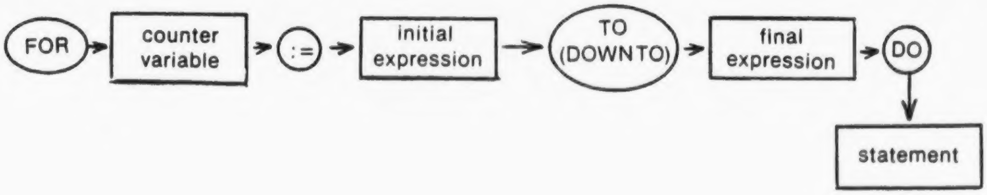
6. How can we use characters as the initial and terminal values of a FOR-DO statement? This program segment, for example, will write out the alphabet:

```
FOR ch := 'A' TO 'Z' DO
  write (ch : 2);
```

7. In all the examples so far, the body of the loop has been a single statement. If the loop body consists of several statements, they must be enclosed with BEGIN-END delimiters.

80

8. This is the syntax diagram of the FOR-DO statement:



If the statement is a compound statement, it must be bracketed with BEGIN-END delimiters. The counter variable must be of the same ordinal type as the initial and final expressions. While the value of the counter variable can be used in the body of the loop, it cannot be altered.

9. Using the FOR-DO loop, we can write a program that reads in and sums ten real numbers:

```
PROGRAM Adder (input, output);
VAR
  sum, num : real;
  X : integer;
BEGIN
  sum := 0;
  FOR X := 1 TO 10 DO
    BEGIN
      writeln ('Enter number');
      readln (num);
      sum := sum + num
    END; (* FOR-DO *)
  writeln ('The sum of your 10 numbers is ', sum : 10 : 2)
END. (* Adder *)
```

10. A FOR-DO loop is also appropriate for a program that finds the sum of a harmonic series: 1 + 1/2 + 1/3 + . . . 1/n:

```
PROGRAM Harmonic (input, output);
VAR
  N, denom : integer;
  hsum : real;
BEGIN
  writeln ('How many terms?');
  readln (N);
  hsum := 0;
  FOR denom := 1 TO N DO
    hsum := hsum + 1 / denom;
  writeln ('The sum of the first ', N : 1, ' terms');
  writeln ('of the series is ', hsum : 10 : 2)
END. (* Harmonic *)
```

11. Nested loops are a powerful programming tool. In a nested loop, one loop is nested inside another. The inner loop must be completed before the outer loop is incremented.

12. What arithmetic expression will be calculated by the following program segment?

```
ans := 1;
FOR power := 1 TO C DO
  FOR exponent := 1 TO B DO
    ans := A * ans;
```

The inner loop will calculate A^B. The outer loop will execute this C times. The final answer will be $(A^B)^C$.

13. The most appropriate way to work with increments not equal to one is to use a WHILE-DO statement.

Summary:

1. How does a FOR-DO loop differ from the REPEAT and WHILE looping structures?
2. In what programming situations is the FOR-DO loop preferable? Why?
3. What is the syntax of the FOR-DO statement?
4. What is nesting?

Exercises:

1. Write programs that will:

   (a) print the alphabet backwards
   (b) sum the consecutive integers between two values entered on input
   (c) find the average of ten numbers entered on input
   (d) produce ten lines of forty *'s each
   (e) list the integers from one to ten, their squares, and their cubes in a three-column table

Topic 17
CASE STATEMENTS

Performance Objectives: Students will be able to:

- explain the syntax and operation of the CASE statement
- use the CASE statement in computer programs
- cite examples of situations in which the CASE statement can be used effectively

Vocabulary: CASE statement, selection expression, error trap

Motivation:

- In college, a student's grade point average (sometimes called an index) is determined by converting a letter grade to quality points. Write a program that will convert letter grades to quality points, using the following values:

Letter Grade	Quality Points
A	4
B	3
C	2
D	1
F	0

- What decisions must be made by the computer program?
  - whether the grade is on A, B, C, D, or F
- Since the program must make more than two decisions, what type of statement must be used?
  - compound IF statement
- Elicit the following program:

```
PROGRAM Grades (input, output);
(* This program will convert letter grades to quality points *)
VAR
  lgrade : char; (* letter grade *)
  qpoints : integer; (* quality points *)
BEGIN (* Grades *)
  writeln ('Please enter the letter grade');
  readln (lgrade);
  IF (lgrade = 'A') OR (lgrade = 'B') OR
     (lgrade = 'C') OR (lgrade = 'D') OR (lgrade = 'F') THEN
    BEGIN (* convert the grades *)
      IF lgrade = 'A' THEN
        qpoints := 4
      ELSE
        IF lgrade = 'B' THEN
          qpoints := 3
        ELSE
          IF lgrade = 'C' THEN
            qpoints := 2
          ELSE
            IF lgrade = 'D' THEN
              qpoints := 1
            ELSE
              qpoints := 0;
      write ('A letter grade of  ', lgrade, ' is equivalent to ');
      writeln (qpoints,' quality points.')
    END
  ELSE
    writeln ('Please enter only grades of A, B, C, D, or F')
END.  (* Grades *)
```

. In the program Grades, multiple decisions are being made. Pascal provides a control structure that allows for multiple decisions.

Aim: How can the CASE statement be used to choose between several alternative actions?

Development:

1. How can the letter grades (lgrade) be converted to quality points (qpoints) using the CASE statement?

   - This is the CASE statement for the conversion:

```
CASE lgrade OF
  'A' : qpoints := 4;
  'B' : qpoints := 3;
  'C' : qpoints := 2;
  'D' : qpoints := 1;
  'F' : qpoints := 0
END; (* CASE lgrade *)
```

2. In this CASE statement, the expression lgrade was selected for evaluation. The value of lgrade, which is a selector expression, will then be compared to each choice in the CASE label list (A, B, C, D, F) to determine which statement will be executed.

3. Which statement will be executed when the selector expression lgrade is evaluated as:

   'A' (qpoints := 4;)
   'C' (qpoints := 2;)
   'F' (qpoints := 0;)

4. After lgrade is evaluated as A, B, C, D, or F and the corresponding statement is executed, where does control pass next?

   - to the first statement after the CASE END

5. How would the CASE statement evaluate an lgrade of 'Z'?

   - In standard Pascal, if the value of lgrade (the selector) is not found in the CASE label list, an error message will be printed and the program will be terminated. In UCSD Pascal, if the value is not found in the CASE label list, no statement will be executed, and control will pass to the first statement following the CASE END.

6. How can you be sure that only grades of A, B, C, D, or F are used?

   - Use an error trap before the CASE statement. Error traps help make programs less sensitive to user errors. They end programs instead of crashing them. Program Grades illustrates the use of an error trap when a user enters a grade other than A, B, C, D, or F.

7. If lgrade is declared as type character, can the CASE label list have integer grades?

   - No, the selector expression and CASE label list must be of the same type.

8. What data types can the selector expression be?

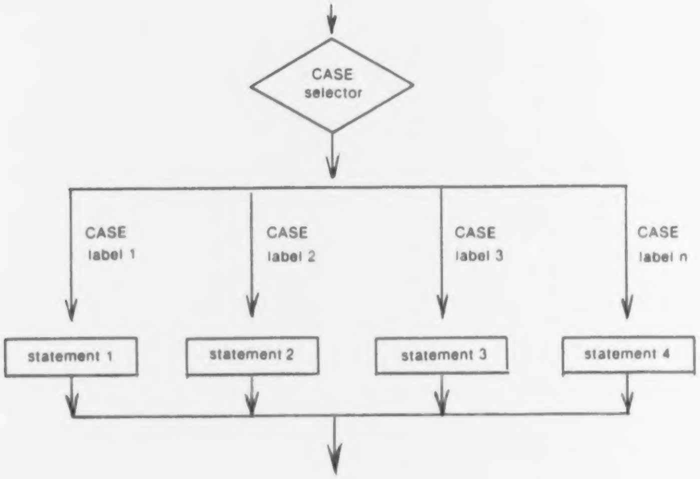
   - The selector expression can be integer, boolean, or character--but not real. That is, the selector expression can be any ordinal-type value. (It can also be any scalar or subrange data type, which will be discussed in a later topic.)

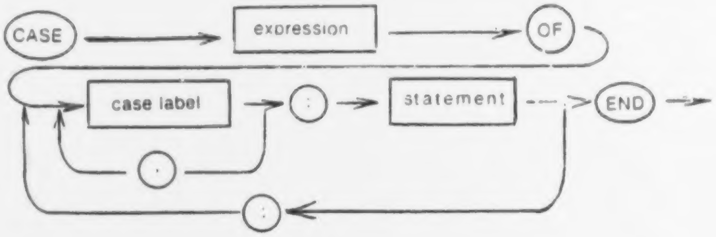
9. How can we alter the program Grades to use the CASE statement?

   - This is one solution:

```
PROGRAM Grades (input, output);
(* This program will demonstrate the conversion of a *)
(* letter grade to quality points using the CASE statement *)
VAR
  lgrade : char; (* letter grade *)
  qpoints : integer; (* quality points *)
BEGIN
  writeln ('Please enter the letter grade');
  readln (lgrade);
  IF (lgrade = 'A') OR (lgrade = 'B') OR
     (lgrade = 'C')OR (lgrade = 'D') OR (lgrade = 'F') THEN
    BEGIN
      CASE lgrade OF
        'A' : qpoints := 4;
        'B' : qpoints := 3;
        'C' : qpoints := 2;
        'D' : qpoints := 1;
        'F' : qpoints := 0
      END; (* CASE lgrade *)
      write ('A letter grade of ', lgrade,' is equivalent to ');
      writeln (qpoints,' quality points')
    END
  ELSE
    writeln ('Please enter only grades of A, B, C, D, or F')
END.  (* Grades *)
```

10. The following flowchart and syntax diagram explain the action of the CASE statement:


CASE selector
CASE label 1
CASE label 2
CASE label 3
CASE label n
statement 1
statement 2
statement 3
statement 4


CASE
expression
OF
case label
:
statement
END
,
;

As these diagrams illustrate, the selector expression is evaluated first. Then the CASE label list is checked. If the appropriate label is found, the corresponding statement or statements are executed and control passes to the next statement following the CASE END.

11. What is the general form of the CASE statement?

```
CASE selector OF
  case label 1 : statement 1;
  case label 2 : statement 2;
  ...
  case label n : statement n
END; (* CASE selector *)
```

. Notice that:

- The CASE label list and corresponding statement are separated by a colon.
- The reserved word END terminates the CASE statement and is not preceded by a semicolon.

12. The following problem illustrates the use of the CASE statement:

A woman is driving her car on the highway. When she drives over 59 mph, her speedometer displays the message "You are speeding." When she drives under 50 mph, her speedometer displays the message "You are driving too slow." When she drives between 50 and 59 mph, no message is displayed.

. Write a CASE statement that displays these messages.

```
CASE speedometer DIV 10 OF
   0, 1, 2, 3, 4 : writeln ('You are driving too slow');
               5 : ;
  6, 7, 8, 9, 10 : writeln ('You are speeding')
END; (* CASE *)
```

. To indicate that no action is taken when the selector expression is evaluated and found to be 5, a semicolon immediately follows the colon to indicate an empty, or null, statement (a statement with no action).

. Notice that more than one CASE label may execute the same action. CASE labels that execute the same action may be grouped together on the same line, provided that they are separated by commas (e.g., 0, 1, 2, 3, 4).

<u>Applications:</u>

1. Rewrite the program Grades so that it converts a letter grade into quality points and then prints one of these messages: Excellent, Very Good, Good, Poor, Failing.

Possible solution:

```
PROGRAM Grades2 (input, output);
VAR
  lgrade : char; (* letter grade *)
  qpoints : integer; (* quality points *)
BEGIN (* Grades2 *)
  writeln ('Please enter the letter grade');
  readln (lgrade);
  IF (lgrade = 'A') OR (lgrade = 'B') OR
     (lgrade = 'C') OR (lgrade = 'D') OR (lgrade = 'F') THEN
    CASE lgrade OF
      'A' : BEGIN
              qpoints := 4;
              writeln ('Excellent')
            END;
      'B' : BEGIN
              qpoints := 3;
              writeln ('Very Good')
            END;
      'C' : BEGIN
              qpoints := 2;
              writeln ('Good')
            END;
      'D' : BEGIN
              qpoints := 1;
              writeln ('Poor')
            END;
      'F' : BEGIN
              qpoints := 0;
              writeln ('Fail')
            END;
    END (* CASE Grades2 *)
  ELSE
    writeln ('Please enter A, B, C, D, or F only')
END.  (* Grades2 *)
```

Summary:

1. For what type of decisions is the CASE statement used?

2. What type of values can the CASE selector expression be?

3. What will happen if the value of the selector expression is not found in the CASE label list?

4. What should be done if no action is to be performed for a particular CASE label?

5. What is meant by an error trap?

Exercises:

1. Write a program that will take an average grade, round it off, and convert it to a letter grade using the following chart:

Average Grade	Letter Grade
90 - 100	A
80 - 89	B
70 - 79	C
60 - 69	D
0 - 59	F

Topic 18
STANDARD FILES: INPUT AND OUTPUT

Performance Objectives: Students will be able to:

- describe the standard input and output files
- insert statements in a program to direct output to the printer
- format output neatly using the write and writeln procedures

Vocabulary: input and output files, rewrite, text, close, fieldwidth parameters, text file (for UCSD Pascal: interactive files, keyboard)

Note: The techniques described in this lesson are not standard. Check your system.

Motivation:

- Write a program called Printit that will display the following prompt: "Do you want to use the printer (1) or console (2)?" Your program should allow the user to input the choice. If the user chooses (1), have the program activate the printer for output; if the user chooses (2), display the output on the CRT.
- Up until now, all the interactive programs we have written used the system files (input, output). These files allowed the user to input data from the keyboard and view the output of data on the CRT.
- In solving the above problem, it is fairly easy to display the prompt, but how can the printer be activated to give a hard copy of a program's output?

Aim: How can the computer use files to direct output to the printer?

Development:

1. Since input and output files are used so frequently in Pascal, in many systems it is not necessary to declare them in the program header. The use of these standard files is assumed, and neglecting to declare them will cause the computer to default to them. In addition, input and output files are special types of predeclared text files.

2. What is a text file? From the phrase, a text file is a file of individual characters.

3. To print a Pascal text file, a printer can be used as the output file. To do this, a file name (any legal identifier) must be declared in the VAR statement as TYPE text:

```
VAR
  identifier : text;
```

4. Just as a secretary must open a file before using it, we must also open the file in the program body before we can use it.

5. The predefined procedure rewrite is commonly used to create and open a new output file. To use the printer as the output file, the identifier in the VAR statement must be identified with the printer:

```
rewrite (identifier, 'printer:');
```

This will associate the printer with the legal identifier.

6. For example, how would we write the VAR and rewrite statements to identify text file F with the printer? (Elicit responses.)

```
VAR
  F : text;

rewrite (F, 'printer:');
```

7. Where in the program does the rewrite statement belong? Since the rewrite statement will open a specific output file, it is placed in the body of the program, preceding the related output statements.

8. When the text file has been opened and the printer has been activated, how can we get the printer to print the lines of text? To obtain output from the printer, the text file identifier must be included in the writeln statements as the first parameter. This will send program output to the file (that is, the printer).

9. For example, how would we write an output statement that would have the printer print "This is a test of the printer"? Use file F which we declared in the earlier VAR statement. (Elicit responses.)

```
writeln (F, 'This is a test of the printer');
```

This writeln statement will write the sentence "This is a test of the printer" to text file F, which has previously been declared, opened, and identified with the printer.

10. After file F has been completed, how can we insure that nothing else is written to it? After a file is opened, it is customary to close it, using the predefined procedure close. For example:

```
close (F)
```

will close the open file F.

11. Although it is customary to use the close command, it is not necessary. Upon termination of a program, the compiler will close all its opened files.

12. Now that we know how to declare a text file, open it, close it, and activate the printer, let's write program Printit.

- You were asked to display a prompt message on the CRT, to have the user input a choice (1 or 2), and to activate the printer or console. What must be declared? (Elicit responses.)
  - We need to declare the text file in the VAR statement:

```
VAR
F : text; (* text file *)
num : integer; (* numerical choice for printer or console *)
```

- How can you open file F and identify F with the printer or console?
  - To do this, 'printer:' and 'console:' must be associated with the file that has previously been declared as a text file.
  - Since 'printer:' and 'console:' are in quotes and each is a sequence of characters interpreted as one item, each is defined as a string. Therefore, the VAR statement will be:

```
VAR
  F : text; (* text file *)
  num: integer; (* numerical choice for printer or console *)
  choice : string; (* printer or console *)
```

- Now the program can be completed:

```
PROGRAM Printit;
VAR
  num : integer; (* numerical choice for printer or console *)
  choice : string; (* choice of printer or console *)
  F : text; (* text file *)
BEGIN
  write ('Do you want to use the printer (1) or console (2)?');
  readln (num);
  IF num = 1 THEN
    choice := 'printer:'
    ELSE
      choice := 'console:';
  rewrite (F, choice);
  writeln (F, 'This is a test of the printer');
  close (F)
END.  (* Printit *)
```

- Notice that the rewrite statement will associate F with choice. The choice will be either the printer or the console.

13. Now that the output can be printed out, how can we format it neatly? As you have learned in an earlier lesson, proper formatting can be done using fieldwidth parameters.

- The statement

```
writeln (20, 30, -40)
```

  will cause the computer to print

```
2030-40
```

- The statement

```
writeln (1.5, 2.5, -3.6)
```

  will cause the computer to print

```
1.50000_2.50000-3.60000
```

- The statement

```
writeln ('one', 'two', 'three')
```

  will cause the computer to print

```
onetwothree
```

14. Proper formatting needs fieldwidth parameters.

- The statement

```
writeln (25 : 10)
               ↑
          fieldwidth parameter
```

  will cause the computer to print

```
________25
```

- The statement

```
writeln (25 : 10, 20 : 5, -3 : 4)
```

  will cause the computer to print

```
________25___20__-3
```

- The statement

  ```
  writeln ('one' : 8, 'two' : 8, 'three' : 8)
  ```

  will cause the computer to print

  ```
  _____one_____two___three
  ```

15. Output with real numbers needs 2 fieldwidth parameters:

    - The first parameter is the width of the field in which the real number should be printed. The second parameter specifies the number of decimal places that will be printed.

    - For example, the statement

      ```
      writeln (2.65 : 6 : 2)
      ```

      will print

      ```
      __2.65
      ```

    - In contrast, the statement

      ```
      writeln (2.65 : 6 : 3)
      ```

      will print

      ```
      _2.650
      ```

      with 3 decimal places.

    - What will be printed by the following writeln statement?

      ```
      writeln (12.35 : 10 : 2, -73.5 : 10 : 2, 100.0 : 10 : 2)
      ```

    Note for UCSD Pascal: Since input and output files are both located on the CRT, the read and write statements display information on the screen. To prevent input data from being seen on the screen we can use this statement:

    ```
    read (keyboard, variable list);
    ```

    The values that are read will not be echoed on the screen. Input, output, and keyboard are called interactive files.

Summary:

1. What are the standard input/output files?
2. What is a text file?
3. How is a text file declared?
4. What is the purpose of a rewrite statement?
5. What is the syntax for the rewrite statement?
6. How does output get directed to the printer using a write statement?
7. How can a text file be closed?

Topic 19
EOF AND EOLN FUNCTIONS

Performance Objectives: Students will be able to:

- state the purpose of the eof and eoln functions
- indicate when the eof and eoln functions take on the values TRUE and FALSE

Vocabulary: eof, eoln

Motivation:

- How can we write a program that will allow a user to input five values and print them? (Elicit responses.)

- This is one possible solution:

```
PROGRAM Print (input, output);
VAR
  loop : integer; (* loop variable *)
  value : real; (* value to be read *)
BEGIN (* Print *)
  FOR loop := 1 TO 5 DO
    BEGIN (* loop *)
      read (value);
      write (value : 5 : 2)
    END (* loop *)
END. (* Print *)
```

- How can we change this program to allow the user to input an undetermined number of values on a line?

  - Use a WHILE loop, or use a REPEAT loop with a sentinel.

- Pascal provides a boolean function called end of line (written eoln) that determines if the last character on a line of data has been read. It also provides a comparable function called end of file (written eof) that determines if the last character in a file has been read.

Aim: How are the eof and eoln functions used with the read and readln procedures to interpret the input of data?

Development:

Note: <eoln> means the eoln marker, which on most systems is the carriage return.

1. Imagine that the following are three lines of data, that the variables A, B, C, D have been declared as integer values, and that <eoln> indicates the eoln marker:

```
5       23      46      < eoln >
10      16      87      < eoln >
3       5       42      < eoln >
```

. Using read and readln, what values will be read from the given data?

```
(a) Read (A, B);          5      23
    Read (C, D)           46     10

(b) Read (A, B);          5      23
    Readln (C, D)         46     10

(c) Readln (A, B);        5      23
    Readln (C, D)         10     16

(d) Readln (A, B);        5      23
    Read (C, D)           10     16
```

2. Although (a) and (b) take on the same values and (c) and (d) take on the same values, each interprets the data differently.

. When read and readln are executed, imagine that an invisible "pointer" points to the next piece of data that will be used. Look at these examples and imagine the pointer:

```
(a) Read (A, B);          5      23      46      < eoln >
    Read (C, D)           10     16      87      < eoln >
                            ↑

(b) Read (A, B)           5      23      46      < eoln >
    Readln (C, D)         10     16      87      < eoln >
                          3      5       42
                         ↑
```

. In (a), the pointer is placed after the 10, making 16 the next piece of data; in (b) the pointer is after 87, making 3 the next piece of data.

. Where is the pointer in (c) and (d)?

```
(c) The pointer           5      23      46      < eoln >
                          10     16      87      < eoln >
                          3      5       42
                         ↑

(d) The pointer           5      23      46      < eoln >
                          10     16      87
                                   ↑
```

3. How does this relate to reading an undetermined number of values, as in our original problem?

- The eoln function can be used as a control in a loop.

4. Which loop(s) can be used with eoln?

   - REPEAT-UNTIL
   - WHILE-DO

5. In the REPEAT loop, the loop is executed as long as the condition is FALSE. In other words, you repeat the process until the boolean condition is TRUE. Eoln is TRUE when there are no more characters to be read on the line; therefore:

```
REPEAT
  body of loop
UNTIL eoln;
```

6. How would you use a REPEAT loop to read an undetermined number of characters and print them out?

```
REPEAT
  read (X);
  write (X)
UNTIL eoln;
```

7. How does this REPEAT loop interpret the data? Suppose the data to be used in this program was the course name computer science. If this data was entered on a data card, the entire string of characters would be stored in the file input. The Pascal system in that particular machine would insert a special symbol (eoln mark) after the last character on that card. In this case, the eoln mark would be inserted after the last e in the string computer science.

   - Upon entering the loop, each character is read and then written. The computer keeps track of each character (or the current position of the pointer). When the next character to be read is the eoln marker, the value of eoln is TRUE, and control is passed to the next line after the loop.

   - If the same data was entered on a terminal, the data would also be stored in the input file. The only difference is in the way the data is entered. On the terminal, the user will type in a line of data. Again, a special symbol will be inserted after the last character. Pressing the ≺return≻ or ≺enter≻ key would signify that one line of data had been entered.

8. The example above used the read procedure. What would happen if readln were used instead?

```
REPEAT
  readln (X);
  write (X)
UNTIL eoln;
```

- If computer science were again the data, an eoln marker would be placed after the e in science. The entire line of data would be stored in file input. Upon execution of the program, the computer would read the first character, c. The readln would now have the computer skip the rest of the line, hence skipping the eoln marker. It would write c and then look for more input.

- Try running this program on your computer and see what happens. On some computers, this will create an infinite loop, while on others the eoln marker will be reached when a blank line of data has been entered (only the eoln marker).

9. What will happen if the data is of type real or integer, as in the following line of data:

```
-3.42     56
```

   - If the variable is of type real or integer, the read command will cause the computer to skip over any blanks or eoln markers until it reaches a character that is part of the number.

10. How can eoln be used with the WHILE loop?

```
WHILE NOT eoln DO
  BEGIN
    read (ch);
    write (ch)
  END;
(where ch is declared as character)
```

   - In the WHILE loop, the body of the loop is executed as long as the condition is TRUE. In other words, when the condition is FALSE, the computer exits the loop.

   - Eoln is TRUE (NOT eoln is FALSE) when the next character to be read is the eoln character. Eoln is FALSE (NOT eoln is TRUE) when the next character to be read is not the eoln character. In some of the previous examples, the eoln and eof have been used with the REPEAT-UNTIL statement for illustrative purposes only. This can create problems because the check in a REPEAT-UNTIL is a post-test. The WHILE-DO statement is a pre-test statement and eliminates the problem.

11. When will data be processed in the WHILE loop?

   - Data will continue to be processed when the next character to be read is NOT the eoln character. Therefore the condition in the WHILE loop is:

```
WHILE NOT eoln DO
  statement;
```

In other words, while you're not at the end of the line, the statement is executed.

12. We have seen that Pascal allows for an undetermined number of characters on one line to be entered into the computer, but can it handle an undetermined number of characters of a file that has many lines?

  - Use end of file (eof). The function eof is TRUE when the end of the file is reached, i.e., when the file position pointer has passed the last character in the file.

13. Eof is TRUE (NOT eof is FALSE) when the next character to be read in the file is the eof character. Eof is FALSE (NOT eof is TRUE) when the next character to be read in the file is NOT the eof character.

14. In a WHILE loop, when should data be processed?

  - Data should be processed while the next character to be read is not the eof character. Therefore, you would use:

```
WHILE NOT eof DO
```

15. How would you write a program that counts the number of words in a file?

. When do you want to process data?

  - When you are not at the end of the file (WHILE NOT eof).
  - When there is still more data on the data card (WHILE NOT eoln)

. Notice that in this program count must be initialized to 1. If

```
x = ' '
```

then count must be incremented. (On the Apple, CNTRL C indicates eof; On the Radio Shack, the clear key indicates eof.)

. This is one solution to the problem:

```
PROGRAM Words (input, output);
VAR
  X : char;
  count : integer;
BEGIN
  count := 1; (* initializes counter; assumes file is not empty *)
  WHILE NOT eof DO
    BEGIN
      WHILE NOT eoln DO
        BEGIN
          read (X);
          IF X = ' ' THEN
            count := count + 1 (* assume words are separated by only
            one blank *)
        END; (* eoln *)
      count := count + 1; (* assumes that there is no blank after
      the last word on a line *)
      readln
    END; (* eof *)
  writeln ('The number of words is ', count)
END.  (* Words *)
```

When we read in the eoln marker as a character, it will be treated as if it was a blank.

Summary:

1. What is the purpose of eof? eoln?
2. When do eof and eoln take on the values TRUE and FALSE?
3. How are read and readln used with eof and eoln?

Exercises:

1. Write a program to calculate the average number of words per sentence in a paragraph. (Hint: A count will be needed to count the number of words. The computer will need to be told that a new word begins when a blank is encountered. A count will also be needed to count the number of sentences. The computer will need to be told that a sentence ends with a period, question mark, or exclamation point.)

Topic 20
STEPWISE REFINEMENT

Performance Objectives: Students will be able to:

- describe a systematic technique for solving problems on the computer
- identify the essential ideas in stepwise refinement
- analyze a problem using stepwise refinement

Vocabulary: stepwise refinement, pseudocode

Motivation:

- Write a program that will input a sequence of positive integers and print out the largest and smallest numbers and their average. Use zero as a sentinel.

Aim: What is a useful technique for organizing a solution to a problem on the computer?

Development:

1. As we get further into this course, the programming problems will get progressively more difficult and complex. In this lesson we will learn some problem solving techniques and then apply them to the motivational problem.

2. By following the five steps listed below, we can solve even the most difficult problems. These steps are based on the concept that we can simplify a problem by dividing it into subproblems that have fewer tasks and are, therefore, easier to manage and code.

   (a) Understand the problem and its data. Clarify the problem, if necessary, so that you clearly understand what the program is to do.

   (b) Decide on input format, variables for input data, output format, variables for output data, and any other constants and variables that may be needed by the program. Determine the type for each variable, and give it a name that indicates its use.

   (c) Make an outline of the tasks to be accomplished in solving the problem. This outline should be written in English using Pascal decision structures (e.g., WHILE, REPEAT-UNTIL, FOR-TO-DO, IF-THEN, CASE) and appropriate indentation wherever applicable. (This is called pseudocode).

(d) Refine the tasks in the outline to simpler subtasks wherever necessary. Tasks that can be directly translated into Pascal statements do not need refinement, but the others should be broken down into several simpler steps. Wherever these subtasks cannot be directly translated into Pascal statements, further levels of refinement will be necessary. Continue this process until all steps can be easily translated into Pascal statements.

(e) Translate the simplified outline into a Pascal program.

3. Using these steps, we can solve the motivation problem:

. What constants do we need?

```
sentinel = 0
```

. What variables do we need?

```
number : integer (* for input *)
largest : integer (* initialize to -maxint *)
smallest : integer (* initialize to maxint *)
count : integer (* indicates how many numbers have been
  entered; initialize to 0 *)
sum : integer (* indicates sum of numbers entered;
  initialize to 0 *).
average : real
```

. How should we outline this problem?

```
Initialize variables
Input number
WHILE number <> sentinel DO
    Update variables
    Input number
Calculate average
Print out results
```

. How should we refine this outline?

```
Initialize largest, smallest, count, and sum
Input number
WHILE number <> sentinel DO
    Update count
    Update sum
    Update largest, if necessary
    Update smallest, if necessary
    Input number
Calculate average
Print out largest, smallest, and average
```

. At this point, we could skip any further refinement and write the program. The amount of refinement needed will depend on the programmer's ability and knowledge of Pascal. To further illustrate stepwise refinement, we will do one more level of refinement for this problem:

```
Set largest to -maxint
Set smallest to maxint
Set count to 0
Set sum to 0
Input number
WHILE number <> sentinel DO
    Set count to count + 1
    Set sum to sum + number
    IF number > largest
        THEN set largest to number
    IF number < smallest
        THEN set smallest to number
    Input number
Set average to sum / count
Print out largest
Print out smallest
Print out average
```

4. This is the Pascal program for the problem. Notice how it follows our refined outline:

```
PROGRAM Maxminavg (input, output);
CONST
  sentinel = 0;
VAR
  number, largest, smallest, count, sum : integer;
  average : real;
BEGIN
  largest := -maxint;
  smallest := maxint;
  count := 0;
  sum := 0;
  writeln ('Enter number - 0 to end');
  readln (number);
  WHILE number <> sentinel DO
    BEGIN
      count := count + 1;
      sum := sum + number;
      IF number > largest
        THEN largest := number;
      IF number < smallest
        THEN smallest := number;
      writeln ('Enter number - 0 to end');
      readln (number)
    END;
  average := sum / count;
  writeln ('Largest number was ', largest);
  writeln ('Smallest number was ', smallest);
  writeln ('Average was ', average : 10 : 2)
END. (* Maxminavg *)
```

5. By following the steps outlined in this lesson, you will be able to solve many difficult problems that you were previously unable to solve. In the next lesson, this same approach will be applied to the business problem of inventory management.

Summary:

1. What are the five steps that can be used to solve a programming problem?

2. What is stepwise refinement?

3. What is a pseudocode?

Exercises:

Write the following programs using stepwise refinement.

1. Write a program that will print out all of the prime numbers between 1 and 100. (Hint: One of your steps should be to determine whether or not a number is a prime. Recall that a prime number is an integer greater than one which only has two divisors, itself and one.)

2. Write a program that will input the numerator and denominator of a fraction and then output the fraction in reduced form. If the result is an improper fraction (i.e., greater than or equal to 1), your program should change the fraction to a reduced mixed number.

Topic 21

INVENTORY MANAGEMENT

Performance Objectives: Students will be able to:

. explain the process by which a company takes inventory
. analyze a major computer business application using stepwise refinement

Vocabulary: inventory

Motivation:

. Write a program that will allow a mail-order computer supply company to keep track of the number of diskettes to be manufactured and shipped. The program should process orders for diskettes and check whether there is sufficient inventory to fill the orders. If the order can be filled, the program should print the message "order shipped" after the order; otherwise, the message "cannot fill order" should be printed.

A sentinel value of zero should be used to indicate that there are no more orders. After all orders have been processed, the program should print out the number of diskettes left in inventory, the total number of diskettes shipped, and the number of additional diskettes that must be manufactured to fill all outstanding orders.

Aim: How can a company use the computer to take inventory of its stock?

Development:

1. We need to clarify the problem. We will have to know how many diskettes are in stock at the beginning of the program (i.e. the opening inventory). As diskettes are shipped out to fill orders, we will have to reduce the current inventory. Whenever we cannot fill an order we will have to increase the number of diskettes on back order. The number of diskettes in each order will be input as an integer value.

2. As the first step in stepwise refinement, let's set up the following constants and variables:

```
sentinel = 0 (* constant *)
order : integer (* number of diskettes in the current order *)
openinv : integer (* number of diskettes in the opening inventory *)
currentinv : integer (* number of diskettes in the current
    inventory *)
backorder : integer (* number of diskettes on back order from orders
    that could not be filled *)
shipped : integer (* number of diskettes shipped *)
manufacture : integer (* number of diskettes to be manufactured.
    This will be equal to the number of diskettes on back order less
    the number of diskettes left in the inventory after all orders
    have been processed, i.e., the closing inventory *)
```

3. How should we outline the problem?

```
Read opening inventory
Initialize currentinv and backorder
Process all orders
Print closing inventory
Compute and print shipped
Compute and print manufacture
```

4. How should we refine this outline?

```
Read openinv
currentinv := openinv
backorder := 0
Read first order
WHILE order <> sentinel DO
    Process current order
    Read next order
Print closing inventory (currentinv)
shipped := openinv - currentinv
Print shipped
IF backorder > 0 THEN
    manufacture := backorder - currentinv
    Print manufacture
```

5. How should we refine this outline further?

```
Read openinv
currentinv := openinv
backorder := 0
Read order
WHILE order <> sentinel DO
    IF order <= currentinv THEN
        Print "Order shipped"
        Reduce currentinv by order
    ELSE
        Print "cannot fill order"
        Increase backorder by order
    Read order
Print closing inventory (currentinv)
shipped := openinv - currentinv
Print shipped
IF backorder > 0 THEN
    manufacture := backorder - currentinv
    Print manufacture
```

6. This is a Pascal program for the problem:

```
PROGRAM Inventory (input, output);
CONST
  sentinel = 0;
VAR
  order, openinv, currentinv, backorder,
  shipped, manufacture : integer;
BEGIN
  writeln ('Enter opening inventory');
  readln (openinv);
  currentinv := openinv;
  backorder := 0;
  writeln ('Enter next order - 0 to end');
  readln (order);
  WHILE order<>sentinel DO
    BEGIN
      IF order<= currentinv THEN
        BEGIN
          writeln ('Order shipped');
          currentinv := currentinv - order
        END
      ELSE
        BEGIN
          writeln ('Cannot fill order');
          backorder := backorder + order;
        END;
      writeln ('Enter next order - 0 to end');
      readln (order)
    END;
  writeln ('Closing inventory is ', currentinv);
  shipped := openinv - currentinv;
  writeln ('number of diskettes shipped is ', shipped);
  IF backorder>0 THEN
    BEGIN
      manufacture := backorder - currentinv;
      writeln ('Number of diskettes to be
      manufactured is ', manufacture)
    END
END.  (* Inventory *)
```

7. This inventory problem shows how the techniques we learned in the last lesson can be used to help solve a business problem. In fact, these same techniques can be used to solve any problem to be done on the computer. These techniques will become more and more useful as our programming problems get increasingly more difficult and involved.

Summary:

1. What is an inventory?

2. In sequence, what tasks must be done to maintain the inventory of a product?

Exercise:

1. Write a program that will keep your checkbook balance. It should be able to handle deposits made to the account, checks written against the account, and service charges imposed on the account.

Topic 22
FUNCTIONS

Performance Objectives: Students will be able to:

- explain the syntax of functions
- describe the operation of functions
- differentiate between formal and actual parameters
- describe a local variable
- discuss how functions interact with the main program

Vocabulary: function declaration, function heading, function designator, formal parameter, actual parameter, calling a function, local variables

Note: This lesson may be broken up into two separate lessons or combined with the next lesson. More exercises using functions will be included in the next lesson, even though they could be done by the students in this lesson.

Motivation:

- Write a Pascal program that will print out the numbers from 1 to 100 along with their square and cube roots. (Elicit responses.)

- This is one solution to the problem:

```
PROGRAM Roots (output);
VAR
  squareroot, cuberoot : real;
  counter : integer;
BEGIN
  FOR counter := 1 to 100 DO
    BEGIN
      squareroot := sqrt (counter);
      cuberoot := exp (ln (counter) / 3);
      writeln (counter : 4, squareroot : 10 : 4, cuberoot : 10 : 4)
    END (* FOR *)
END. (* Roots *)
```

- Notice that we used a predefined (standard) function sqrt to calculate the square root, but Pascal does not have a function for calculating cube roots. In BASIC we have an operation symbol for exponentiation to get a cube root (i.e., x↑(1 / 3) or x↑.3333), but Pascal does not have an operation symbol or function for exponentiation.

Aim: How can we create our own functions?

Development:

1. Pascal does have a feature which will enable us to define new functions above and beyond the predefined functions learned previously. Therefore, to calculate the same type of value, like cube root, repeatedly in a program we could define a function to calculate the value. The function declaration would look like this:

```
FUNCTION Cube (number : real) : real;
BEGIN
  Cube := exp (ln (number) / 3)
END; (* Cube *)
```

2. The above four-line segment is called a *function declaration*. It declares (defines) the function. The first line of the function declaration is called the *function heading*, and this is very similar to a program heading. The first word

```
FUNCTION
```

is a reserved word that tells the compiler that a function declaration will follow. The next word

```
Cube
```

is the name of the function, just like *sqrt* is the name of the predefined square root function. The name of the function, *Cube* in this case, is also referred to as the *function identifier*. The next term

```
(number : real)
```

is a list of the function's *formal parameters* and their corresponding types. In this case, we have just one formal parameter, named *number*, which is of type real. This means that when we evaluate the function *Cube*, we must give a real value as an argument to our function, just as the sqrt function had to have an argument of type real or integer. The last item in the function heading

```
: real
```

indicates the type of result the function will be assigned. In this case, the function will be assigned a real value. It is important to note that a function returns one value.

The next part of our function declaration is called the *function body*, and this is like the program body of a program. Our function has a one statement function body. Somewhere in our function body, we must assign a value to our function. In this example, our function Cube gets assigned the value

```
exp (ln (number) / 3)
```

where number is our real argument.

3. This is the general form of a function declaration:

```
FUNCTION Function identifier (formal parameter list) : result type;
local declaration section
BEGIN
  function body
END; (* Function *)
```

   We have discussed everything in the function declaration section except for the local declaration. This will be illustrated in one of the following examples and most of our next lesson will be devoted to local declarations.

4. Our formal parameter list consisted of just one parameter. A function can have many parameters of the same or different types. Commas are used to separate the names of several parameters of the same type, while semicolons are used to separate the various types specified, e.g.,

   (X : integer; Y : integer; Z : integer) and (X, Y, Z : integer) both indicate three formal parameters of type integer.

   (X, Y : integer; Z : real) indicates three formal parameters where the first two are of type integer and the third is of type real.

5. The result type of a function must be one of the four standard data types (boolean, integer, real, or char) or a previously defined scalar or subrange type. Scalars and subrange types will be covered in subsequent lessons.

6. If there are no formal parameters, the formal parameter list and parentheses should be omitted.

7. At least one statement must be contained in the function body which assigns a value to the function. We can have more than one such statement, as the following example illustrates:

```
FUNCTION Sign (value : real) : char;
BEGIN
  IF value>= 0 THEN
    Sign := '+'
  ELSE
    Sign := '-'
END; (* Sign *)
```

   Although there are two statements that can assign a value to <u>Sign</u>, it should be obvious that one and only one of these statements will get executed. Good programming style and most compilers require that you assign a value to the function once and only once in the function body. That is, even though the function name gets assigned a value like a variable, <u>do not</u> use the function name as a variable that changes value. Assign it a value only once in your program. If you need a variable that changes values, use another variable and set the function name equal to this variable once it has the correct value. The next example will illustrate this.

8. We may declare variables for use in our functions. These variables are called local variables and can only be used by our function. (Local variables will be discussed in greater detail in the next lesson.) The program below, which finds the largest of three integers, uses local variables:

```
FUNCTION Largest (first, second, third: integer): integer;
VAR
  sublarge : integer;
BEGIN
  IF first > second THEN
    sublarge := first
  ELSE
    sublarge := second;
  IF third > sublarge THEN
    sublarge := third;
  Largest := sublarge
END; (* Largest *)
```

Note that sublarge was declared as a local variable for use in the function.

9. To include a function in a program, we insert the function declaration after the variable declarations of the program and before the program body. For example, our motivational problem would be rewritten as:

```
PROGRAM Roots (output);
VAR
  squareroot, cuberoot: real;
  counter : integer;

FUNCTION Cube (number : real) : real;
BEGIN
  Cube := exp (ln (number) / 3)
END; (* Cube *)

BEGIN (* main program *)
  FOR counter := 1 TO 100 DO
    BEGIN
      squareroot := sqrt (counter);
      cuberoot := Cube (counter);
      writeln (counter : 4, squareroot : 10 : 4, cuberoot : 10 : 4)
    END (* FOR *)
END. (* Roots *)
```

10. Notice the statement

```
cuberoot := Cube (counter)
```

in the main program. This is an example of a function call. Calling a function is another way to say that we are requesting our function to be executed. The function gets called (referenced) by the function designator, Cube (counter), which consists of the function name and a list of arguments called actual parameters. These actual parameters are the actual values that will be assigned to our formal parameters in the function. These actual parameters are like the arguments in our predefined functions. They are the actual values that will determine the value of our function.

11. Whenever a call to a function is encountered in a program, control is transferred from the calling program to the function referenced (called). The function uses the argument(s), and when the function computation is complete, the result is returned and control is tranferred back to the calling program at the point of call.

12. An actual parameter can be a literal, constant, variable, or expression of the correct type. It may even be a function, as long as the result type is correct. The actual parameter values will be assigned to the corresponding formal parameters. This correspondence is determined by order of occurrence (i.e., the first formal parameter gets assigned the value of the first actual parameter, etc.). Of course, the number of actual parameters and their corresponding types must agree with those of the formal parameters.

13. Remember that it is the value(s) of the actual parameter(s) that are used to calculate the value of the function. When we are writing the function declaration, we do not know the actual values of the parameters so we use the formal parameters as place holders (or variable names) for the actual parameters.

Summary:

1. What are the names of the various parts of a function declaration?

2. What are the names of the various parts of a function designator?

3. What is the difference between actual and formal parameters?

4. What is a local variable?

5. How does a function call affect program control?

Exercises:

1. Write a real valued function (i.e., a function whose result type is real) that will take on the value of the average (mean) of its two real valued parameters.

2. Write an exponentiation function for nonnegative integral exponents. That is, write a real valued function called raise to the power. You should have a real parameter called base and an integer parameter called exponent (you may assume that this parameter is nonnegative). Your function should return the value of base raised to the power exponent. (Assumptions about parameters will be discussed in the next lesson).

3. Given the function heading:

```
FUNCTION Test: (A, B : integer; C : real) : integer;
```

tell whether the following function designators in the main program are valid or invalid and why, given the following program constants and variables (you may assume that the variables have been given values):

```
CONST
  pi = 3.14159;
  one = 1;
VAR
  I, J, K : integer;
  D, F : real;
```

```
(a) Test (one, 7, pi)                        (b) Test (I, J, D)
(c) Test (I, J, K)                           (d) Test (K, D, F)
(e) Test (I, I, C)                           (f) Test (I, L, F)
(g) Test (I, trunc (D), sqrt (K))            (h) Test (D, I, J)
(i) Test (I * J, I / J, sqrt (K))            (j) Test (I, I, I)
(k) Test (J MOD I, Test (I, 4, 3.2), 9.7)
```

Topic 23

LOCAL VARIABLES AND SIDE EFFECTS

Performance Objectives: Students will be able to:

- tell the difference between local and global variables
- explain how side effects can be avoided in functions
- explain how functions can be nested

Vocabulary: global variable, local variable, side effect, scope, value parameter.

Motivation:

- Write an integer valued function that will return the lowest common multiple (lcm) of two integers, e.g., lcm (6, 8) = 24, lcm (2, 6) = 6, and lcm (4, 5) = 20. (You may assume that these integer parameters are positive.) (Hint: Write down the steps you would use to calculate lcm (24, 45) without using prime decomposition. Then write a program that will do the same steps.)

- One solution to our problem would be to find the lowest common multiple by the technique you were taught in elementary school. That is, take the larger integer, and see if it is the lowest common multiple by testing whether the smaller number divides into it evenly. If so, you are done; if not, double the larger integer and try again. If you still have not found a solution, triple the larger number. Since the product of the smaller and larger numbers will yield a common multiple, you know that your search will eventually yield a result. This is a function that follows this approach:

```
FUNCTION Lcm (first, second : integer) : integer;
(* This function calculates the lowest common multiple of *)
(* first and second.  Assume that first and second are positive. *)
VAR
  large, small, testvalue : integer;
BEGIN
  IF first > second THEN
    BEGIN
     large := first;
     small := second
    END
  ELSE
    BEGIN
      large := second;
      small := first
    END;
  testvalue := large;
  WHILE testvalue MOD small <> 0 DO
    testvalue := testvalue + large;
  Lcm := testvalue
END; (* Lcm *)
```

Note: It is a very useful technique to work out a difficult problem manually before attempting to program it.

Aim: How can local variables be used in functions to control side effects?

Development:

1. Recall that the variables large, small, and testvalue are called local variables. These variables are declared by the function and exist only while the function is being executed. You may refer to these variables only within the function; otherwise you will create a run-time error. Also, if you call the function once and then leave the function upon completion, the local variables are no longer accessible. Any subsequent calls to the same function will cause the creation of new local variables. Though these local variables will have the same names as the original local variables, they may not retain the values they had at the time you completed the previous call to that function. We say that the function declaration is the scope of these local variables. The function declaration is also the scope of the formal parameters, any locally defined constants, and as we shall see, any other local definitions, such as other functions. This means that you may only access these items while the program is executing within the function declaration of the function in which the variables are declared.

2. Variables declared in the program declarations are referred to as global variables. These variables are not only available while in the program body, but in most cases are also available while executing functions. First, let us consider the case where we have a variable named number that is declared both in the program as a global variable and in a function within the program as a local variable (or formal parameter). Any reference to number within the program body (even as part of an actual parameter for a function designator which calls our function) will use the current value of the global variable number. While within the function declaration, any reference to number will use the current value of the local variable. In this case, the global variable number is not available to the function. Keep in mind there are two distinct variables called number, one in the main program and a different one, with the same name, in the function.

3. Where number is declared only as a global variable, any reference to it while in the program body will use the value of this variable. While in the function declaration, any reference to number will look first at the local variables, local constants, and formal parameters. If number is not declared as one of these, the function will go back and use any globally defined number. Of course, if we had no globally defined number, a run-time error would occur. Finding number declared globally but not declared locally could lead to some undesirable occurrences, called side effects. Using the global value of number for number in

your function may or may not be an erroneous action. But up until now, a function only changed the value of local variables, formal parameters, and the resulting value of the function. At no time did we ever change the value of any variables that were globally defined. Even when global variables were used as actual parameters, only their current values were assigned to the corresponding formal parameters. Any change of the formal parameter had no effect on the actual parameter. The actual parameter was only used to initialize the formal parameter. Such parameters are referred to as value parameters. Let us consider what happens if the assignment statement

```
number := number + 1;
```

is executed in the function previously mentioned. Since there is no local variable number, the global number will get incremented by 1. This is called a side effect of the function, since the function has changed a variable beyond its function declaration. Whether or not you want to change this global variable, it is not considered good programming for a function to produce side effects. Changes in global variables are achieved by another type of parameter, which will be considered in a future lesson.

4. It is legal to have nested function calls. This occurs when a function is used as an actual parameter in its own function designator. Of course, type compatibility must be observed. Therefore, if we want to use our Lcm function to calculate the lowest common denominator of 8, 12, and 20, we can use

   ```
   Lcm (Lcm (8, 12), 20)
   ```

   instead of having two separate calls to Lcm.

5. The local declaration section of a function can contain any declaration that the global declaration section can contain--even a function. That is, one function can locally declare another function for its use. This second function can be used only by the first function and not by the program body. You can use this structure if you have a function that is not needed by the main program body, but which will simplify the writing of a desired function. For example, let us assume that we have previously written our Lcm function. Now we are writing a program that needs a greatest common factor (gcf) function. Furthermore, assume that we are aware of the mathematical equation which states that the greatest common factor of two integers is equal to the product of the two integers divided by the lowest common multiple of the two integers. We can then write the greatest common factor function as follows:

```
FUNCTION Gcf (int1, int2 : integer) : integer;
(* Gcf calculates the greatest common factor of two positive *)
(* integers. *)

FUNCTION Lcm (first, second : integer) : integer;
(* This is a local function that calculates the lowest common *)
(* multiple of two positive integers. *)
VAR
  large, small, testvalue : integer;
BEGIN
  IF first >second THEN
    BEGIN
      large := first;
      small := second
    END
  ELSE
    BEGIN
      large := second;
      small := first
    END;
  testvalue := large;
  WHILE testvalue MOD small<>0 DO
    testvalue := testvalue + large;
  Lcm := testvalue
END; (* Lcm *)

BEGIN (* Gcf *)
  Gcf := int1 * int2 DIV Lcm (int1, int2)
END; (* Gcf *)
```

The main program body can call only the Gcf function and not the Lcm function. Once the Gcf function begins execution, it can call the Lcm function. This is an example of a nested function. It is also possible for the Lcm function to contain a locally declared function for its use.

6. In the Gcf and Lcm functions we restricted our parameters to positive integers. We usually make any necessary assumptions about the parameters while writing the function and then test for these conditions before we call the function. We could test for these conditions in the function, but we would have to return a dummy value for the function and then test for this dummy value before we used the value returned by the function. This latter approach is not preferred. It is a good idea to include any assumptions about the parameters of a function in the opening comment (as we did in the Gcf and Lcm functions).

7. Boolean functions can help us write more concise and readable programs. They are especially helpful when dealing with complex conditions. Rather than writing these conditions over and over again in statements requiring conditional expressions, we can replace the conditions with a boolean function. For example, the function Valid can be used to test whether a number is equal to 35 or between 20 and 30 inclusive, if we define Valid as follows:

```
FUNCTION Valid (number : integer) : boolean;
(* Valid is true if number is 35 or between 20 and 30 inclusive. *)
BEGIN
  Valid := ((number >= 20) AND (number <= 30)) OR (NUMBER = 35)
END; (* Valid *)
```

Summary:

1. What is a local variable?

2. What is a global variable?

3. What is a side effect?

4. What is a nested function?

Exercises:

1. Write a boolean function that will determine whether a positive integer is prime or not.

2. Write a direct integer function that will calculate the greatest common factor of two positive integers (i.e., do not use the lowest common multiple function).

3. Using the solution to Exercise 1, write a nested integer function that will return the next largest prime number for a given positive integer (not necessarily a prime).

4. Write an integer function that will return the number of divisors (counting 1 and itself) of the positive integer parameter.

5. Write a boolean function that will determine if its nonnegative parameter can be written as the sum of the squares of two integers. Include this in a program that will print out the values of all such numbers from 0 to 100.

6. Write a real function that will return the value of the distance between two points expressed as ordered pairs. Use the real parameters X1, Y1, X2, and Y2. Recall that:

$$\text{distance} = \sqrt{(X_1 - X_2)^2 + (Y_1 - Y_2)^2}$$

7. Write a boolean function that will determine whether three ordered pairs of integers [(X1, Y1), (X2, Y2), (X3, Y3)] are colinear. (Note: Be careful to use a test that is valid for all possible integers.)

Topic 24

PROCEDURES WITHOUT PARAMETERS

Performance Objectives: Students will be able to:

- describe the syntax of procedures without parameters
- explain how procedures interact with the main program
- state the reasons for creating procedures
- explain the difference between a function and a procedure
- write a procedure without parameters

Vocabulary: procedure, procedure declaration, procedure heading, procedure identifier, procedure statement, calling a procedure

Motivation:

- Have students write a program that will read two integers and then print out the larger number followed by the smaller number. (Elicit responses.)

- Students should be able to solve the problem using a direct solution. After they have gone over the solution, put up the following program:

```
PROGRAM Largernumber (input, output);
(* Program reads in two integers and *)
(* prints out larger number first. *)
VAR
  first, second, large, small : integer;

PROCEDURE Getthenumbers;
BEGIN
  writeln ('This program prints the larger number first.');
  writeln;
  writeln ('Enter two integers.');
  readln (first, second)
END; (* Getthenumbers *)

PROCEDURE Findlargernumber;
BEGIN
  IF first > second THEN
    BEGIN
      large := first;
      small := second
    END
  ELSE
    BEGIN
      large := second;
      small := first
    END
END; (* Findlargernumber *)
```

```
PROCEDURE Printresults;
BEGIN
  writeln ('Larger number is ', large);
  writeln ('Smaller number is ', small)
END; (* Printresults *)

BEGIN (* main program *)
  Getthenumbers;
  Findlargernumber;
  Printresults
END. (* Printresults *)
```

Notes:

- This solution is much longer than the direct solution, but it illustrates a very useful structure available in Pascal. The structure is called a procedure.
- Although the procedures Getthenumbers, Findlargernumber, and Printresults use global variables for illustrative purposes, in general this is not a good practice because it may cause undesireable side effects. The proper technique is to use variable parameters to pass the global variables (see Topic 25).

Aim: How can procedures be used to implement subprograms?

Development:

1. A procedure is a subprogram or subroutine, much like a subroutine in BASIC. Pascal allows us to assign a name to a set of Pascal statements. When we use this name, the associated Pascal statements will be executed. In the program Largernumber, we have a procedure called Getthenumbers, which is associated with four Pascal statements (three writeln statements and one readln statement). As you can see, procedures are declared in the same manner as functions. The first line of the declaration is called the procedure heading. It contains the reserved word PROCEDURE followed by the procedure identifier (the name of the procedure). In the next few lessons, we will see that we can also have parameters, but we will not use any parameters in this lesson. Also, we can have local declarations, as we did with functions. The statements associated with the procedure are called the procedure body. Together, the procedure heading, local declarations, and procedure body make up the procedure declaration.

2. We call (or invoke) a procedure by using the procedure identifier in a procedure statement. For example, in the main program we call the procedure named Getthenumbers by using its procedure identifier as a procedure statement. When the procedure statement (procedure call) is executed, local variables are set up and the statements in the procedure body are executed. After all the statements in the procedure body are executed, the program resumes by executing the statement immediately following the procedure statement (procedure call). This structure is very similar to the GOSUB and RETURN of BASIC. Note that we have been using the procedure identifier both to declare and to call a procedure.

3. As with functions, procedure declarations follow global variables (VAR) and precede the main body of the program. There is no priority among procedures and functions (i.e., they can be declared in any order as long as they all follow the global variables and precede the main body of the program). But the order in which they are declared will affect their ability to use other procedures and/or functions. In particular, a function or procedure may use any function and/or procedure that is declared prior to it.

4. Although procedures and functions appear to be similar, there are major differences between the two structures. A function takes on a particular value of a particular type and may therefore be viewed as a variable. A procedure does not have a value. A procedure is associated with a set of statements, and it may therefore be viewed as a statement.

5. A procedure may contain local procedures and/or functions. Such procedures are known as nested procedures.

6. There are many advantages to using procedures, and these advantages outweigh the slight increase in the length of the program. Procedures can be used to break up a complicated task into several smaller tasks. Our example did not fully illustrate this advantage, since the given task was not very complicated. Breaking a complicated task into tasks that are smaller and easier to solve is similar to BASIC. (This top-down approach to programming will be discussed further in a subsequent lesson.)

   Breaking down a program into procedures and then having the main program consist mainly of calls to these procedures tends to make the program more readable and more free of errors. It also makes the program easier to debug and easier to test. We can test individual procedures separately, just as we tested subroutines in BASIC.

7. As in BASIC, a procedure is written only once but may be called many times from many different locations within the program. This avoids having to write the same set of statements over and over again.

Summary:

1. What are the names of the various parts of a procedure declaration?
2. What happens when a program executes a procedure statement?
3. What are the advantages of using procedures?
4. What is the difference between a function and a procedure?
5. How does the main program call a procedure?

Exercises:

1. Write a procedure that will ask for three integers and read them in as first, second, and third. Do not declare these variables in your procedure; assume that they have been declared globally.

2. Write a procedure that will print out the alphabet from A to Z. (Hint: Use a loop and a local variable.)

3. Write a procedure that will count from 1 to 1,000. There should be no output for this procedure. This procedure will just slow down output like subroutines you wrote in BASIC.

4. Write a procedure that will skip 15 lines.

Topic 25
PROCEDURES WITH PARAMETERS

Performance Objectives: Students will be able to:

- write procedures with parameters
- explain the difference between actual and formal parameters
- explain the difference between value and variable parameters
- contrast the effects of transmitting information to a procedure by value and by variable parameters

Vocabulary: value parameter, variable parameter, formal parameter, actual parameter

Motivation:

- Write a procedure that will print out a row of 20 asterisks.

- This is one solution to the problem:

```
PROCEDURE Print20asterisks;
CONST
  width = 20;
VAR
  count : integer;
BEGIN
  FOR count := 1 TO width DO
    write ('*');
  writeln
END; (* Print20asterisks *)
```

Aim: How can a procedure communicate effectively with the main program?

Development:

1. The procedure we just wrote can be changed to print out a varying number of asterisks. We can do this easily by using a parameter (just as we did with functions). The parameter will specify the number of asterisks to be printed:

```
PROCEDURE Printasterisks (width : integer);
VAR
  count : integer;
BEGIN
  FOR count := 1 TO width DO
    write ('*');
  writeln
END; (* Printasterisks *)
```

2. The procedure Printasterisks again illustrates the use of a *formal parameter*, in this case, *width*. A formal parameter is a variable listed in the heading of the procedure. When we subsequently call the procedure we will specify an *actual parameter*. The formal parameter will initially take on the *value* of the actual parameter. For this reason we call these parameters *value parameters*. Thus, a procedure call of

```
printasterisks (10)
```

will print out a row of 10 asterisks.

3. We can further modify our procedure to print out a specific number of specified characters. For example:

```
PROCEDURE Printchar (ch : char; width : integer);
VAR
  count : integer;
BEGIN
  FOR count := 1 TO width DO
    write (ch);
  writeln
END; (* Printchar *)
```

This procedure has two value parameters, *ch* and *width*. In the procedure call we will now have to provide two values. For example:

```
Printchar ('.', 30)
```

will print out a row of 30 periods.

4. By nesting procedures or by declaring Printchar first, we can write a procedure using Printchar that will print out triangular shaped figures. These figures can be made up of any character and can have any specified depth. The results will look like this:

```
.                *                 $
..               **                $$
      or         ***      or       $$$
                                   $$$$
                                   $$$$$
```

```
PROCEDURE Triangle (ch : char; depth : integer);
VAR
  count : integer;
BEGIN
  FOR count := 1 TO depth DO
    Printchar (ch, count)
END; (* Triangle *)
```

6. The Triangle procedure will work as long as the Printchar procedure is declared before or within it. Note that the double use of the variables count and ch is perfectly legal. (You may want to change these to different names if you find it too confusing.) We can get our three triangles printed with the following calls:

```
Triangle ('.', 2);
Triangle ('*', 3);
Triangle ('$', 5);
```

7. Thus far, whenever we have called a procedure, we have provided a value (through the actual parameter) for each formal parameter. The formal parameters get initialized to the values of the corresponding actual parameters and then behave like local variables. Any change made to the value of a formal parameter has no effect on the actual parameter associated with it. This type of procedure is called a value parameter.

   Pascal has another type of parameter, called a variable parameter, which behaves in a completely different manner. With a variable parameter, we associate a variable (through the actual parameter) with each formal parameter. Any change made to the value of a formal variable parameter will also change the value of the variable which is given as the actual parameter.

   Variable parameters allow two-way communication between the program and the procedure being called. When we used value parameters, we could only send a value to the called procedure. There was no way for the procedure to pass any information back to the calling program. With variable parameters we can change the value of a variable in the calling program, thereby passing information to that program.

8. The following program illustrates this two-way communication:

```
PROGRAM Parameters (input, output);
VAR
  A, B, C : integer;

PROCEDURE Change (X : integer; VAR Y : integer);
BEGIN
  X := X + 2;
  Y := Y + 3;
  writeln (X, Y)
END;

BEGIN (* main program *)
  A := 2; B := 3; C := 5;
  writeln (A, B, C);
  Change (B + C, A);
  writeln (A, B, C)
END.  (* main program *)
```

The procedure Change has two parameters: X is our usual value parameter and Y is our new variable parameter. Notice that a variable parameter is preceded by the reserved word VAR. When the procedure is called, we must provide an integer value for X and an integer variable to associate with Y. Our program provides the value of 8 (i.e., B + C) for X and associates the variable A with Y. When the procedure executes, X (which has the initial value of 8) gets incremented by 2 to take on the value of 10. Since Y is associated with the variable A, when Y gets incremented by 3 it is really A that gets incremented by 3. As far as the computer is concerned, there is no separate variable called Y. When Y is referenced, the computer knows to look at the variable A. In our procedure, A gets incremented by 3 to take on the value of 5 (i.e., 2 + 3), and since Y is still associated with A, it too has a value of 5. When we return to the calling program, we will find that A has a new value. The output to the previous program will be:

```
2 3 5
10 5
5 3 5
```

Notice in the last line of output that A (which was associated with Y) now has a value of 5 (not 2). By changing the value of a variable in the calling program, the called procedure can now send information back to the calling program.

9. The actual parameter for a variable parameter must be a variable. It cannot be a literal constant, declared constant, or expression.

pplication:

1. Determine the output for the following program:

```
PROGRAM Problem5 (input, output);
CONST
  pi = 3.14;
VAR
  A, B : real; C : integer;

PROCEDURE Messup (VAR D : real; E : real);
VAR
  F : real;
BEGIN
  F := D * pi;
  D := 2 * F;
  E := 10 * D;
  writeln (D : 6 : 1, E : 6 : 1, F : 6 : 1)
END; (* Messup *)

BEGIN (* main program *)
  A := 3.0; B := 10.0; C := 5;
  writeln (A : 6 : 1, B : 6 : 1, C : 6);
  Messup (B, C);
  writeln (A : 6 : 1, B : 6 : 1, C : 6)
END. (* main program *)
```

Answer:

```
___3.0__10.0_____5
__62.8_628.0__31.4
___3.0__62.8_____5
```

Summary:

1. What is an actual parameter?
2. What is a formal parameter?
3. What is a value parameter?
4. What is a variable parameter?
5. What type(s) of actual parameters are permitted for a value parameter?
6. What type(s) of actual parameters are permitted for a variable parameter?
7. How can information be sent from a procedure back to a program?

Exercises:

1. Write a procedure called Rectangle that has two parameters called *length* and *width*. The procedure should calculate and print the area and perimeter of a rectangle with the given value of length and width.

2. Write a procedure called Numberofodds that has a parameter called N. The procedure should print out the first N odd numbers.

3. Write a procedure called Fromuptostep that has three parameters called *from*, *upto*, and *step*. The procedure should print out all integers starting at *from*, ending at *upto*, and incrementing by *step*. Do not exceed *upto*. Stop before it, if necessary. (Assume that *upto* is greater than or equal to *from* and that *step* is positive.)

4. Redo Exercise 3 so that it can handle both positive and negative values of *step*. Print out an error message if the printout is infinite (i.e. will never reach *upto*).

Topic 26

PARAMETERS IN PROCEDURES AND FUNCTIONS

Performance Objectives: Students will be able to:

- explain when a parameter should be passed by value or passed by reference
- list reasons for using parameters in procedures and functions

Vocabulary: pass by value, pass by reference, portability of functions and procedures

Motivation:

- Write a procedure called Getnumber that will first ask the user to enter a number and then input an integer, called number, that is to be used in the main program.

- This is one solution to the problem:

```
PROCEDURE Getnumber (VAR number : integer);
BEGIN
  writeln ('Enter an integer');
  readln (number)
END; (* Getnumber *)
```

Aim: How can we decide what type of parameter to use with a procedure or function?

Development:

1. In writing the procedure Getnumber we needed to decide how to write the declaration of the variable number. Should it have been declared as a local variable, global variable, value parameter, or variable parameter? This is usually the most difficult decision encountered with variables. Even though more than one choice can sometimes be made, there usually is only one that is preferable. We will soon see that in writing the procedure Getnumber the best choice was a variable parameter.

2. A local variable is used when a variable is only going to be needed within a procedure or function. Recall that once a procedure or function is completed, the local variables within it no longer exist. Their values are not available to the main program.

3. A global variable is used when a variable is needed in the main program. Pascal allows these variables to be used within procedures and functions. (This will be discussed further in the next topic.) But good programming style recommends the use of variable parameters whenever a global variable will be referred to in a procedure or function. In general, a global variable should not be used in a procedure.

4. A value parameter is used when a procedure or function needs a value from the main program or calling procedure. Value parameters provide copies of values that can be used by procedures and functions. Recall that changes made to the formal parameter (i.e., the copy) have no effect on the actual parameter. Also, the values of formal parameters are not available to the main program or calling routine. Therefore, value parameters provide a one-way communication from the main program or calling procedure to the called procedure.

5. A variable parameter is used when a procedure or function needs to pass information to the main program or calling procedure. It also allows the calling procedure to pass information to the called procedure. Thus, a two-way communication system is provided. Remember that changes made to the formal parameter variable also change the corresponding actual parameter variable.

6. When we use value parameters, we say the information is passed by value. When we use variable parameters, we say the information is passed by reference (or passed by address, or passed by location).

7. Remember that parameters are only used when we have to communicate data between the calling procedure and the called procedure. Also, if only one value has to be passed back to the calling procedure we can often use a function instead of a procedure with a variable parameter. But, whenever two or more values must be passed back to the calling procedure, variable parameters must be used to maintain good programming style.

8. We use parameters to avoid side effects and to make the procedures we write more flexible and portable. By declaring variable parameters whenever a global variable is capable of being changed, we tend to be more aware of possible changes in variables and thus will usually avoid undesirable side effects. Also, by using formal parameters and allowing the caller to associate actual parameter variables with these parameters, we are making the procedures more portable. Now a calling procedure can invoke a called procedure several times, with different variables each time. Finally, the person writing a procedure can do so without knowing the names of the global variables. The programmer can make up names for the formal parameters and worry later about the actual parameters. Thus, several people can work together more easily on a program, and they can test their procedures individually. This also allows a procedure written once to be used in other programs, making it more portable.

Summary:

1. When do we use local variables?

2. When do we use global variables?

3. When do we use value parameters?

4. When do we use variable parameters?

5. What does it mean to pass by value?

6. What does it mean to pass by reference?

7. What are some of the advantages of using parameters?

Exercises:

1. Write a procedure that will input three integers. These three integers are to be used in the main program.

2. Write a procedure that will interchange the values of two real globally-defined variables.

3. Write a procedure that will calculate and print out the greatest common factor (gcf) of two integer parameters. (Note: The value of the gcf will not be needed once it is printed out.)

4. Write a procedure that will reduce fractions that have their numerators and denominators declared globally. The procedure should replace the unreduced values by their reduced values. (Note: Improper fractions are acceptable.)

5. Write a procedure (not a function) that will calculate the larger of two real numbers. This larger value will be needed by the main program.

6. Rewrite the procedure Getnumber as a function.

Topic 27
SCOPE OF AN IDENTIFIER

Performance Objectives: Students will be able to:

- define the scope of an identifier
- describe the scope of identifiers in subprograms
- explain the relationships between the scope of an identifier, side-effects, and parameters
- contrast domain of accessibility and existence
- write nested procedures and functions

Vocabulary: global identifiers, local identifiers, nonlocal identifiers, scope, block

Motivation:

- What will be the output of the following program?

```
PROGRAM Doublex (input, output);
VAR
  X : integer;

PROCEDURE Seeingdouble;
VAR
  X : integer;
BEGIN
  X := 2;
  writeln (X);
  X := X + 1
END; (* Seeingdouble *)

BEGIN (* main program *)
  X := 1;
  writeln (X);
  Seeingdouble;
  writeln (X)
END. (* Doublex *)
```

- The output will be:

```
1
2
1
```

Aim: What is the scope of an identifier?

Development:

1. In the program Doublex there are two different identifiers with the name X; one is global and the other is local to the procedure Seeingdouble. While the procedure is being executed, the local variable X is used whenever X is referenced. Since this local variable exists only while the procedure executes, the main program uses the global variable X.

2. The scope of an identifier is the portion of the program in which the identifier can be used. This portion of the program is called a block. A block consists of three parts: the heading, the declarations, and the statements. (Note that many authors do not treat the heading as part of the block.) In Pascal, blocks appear as programs, procedures, and functions. A variable is available only within the block in which it is declared. Therefore, a variable declared in the declaration section of a program is available throughout the entire program. A variable declared in the declaration section of a procedure is only available within that procedure, or within any of its subprocedures. Procedures and functions are also declared within blocks. In addition to its local variables, a procedure or function may use variables that are declared in the declaration section of any block in which it is nested. Furthermore, procedures and functions may also use variables that are declared in or available to the block that contains its block.

3. Note: In this and following examples, many portions of the program, procedures, and functions have been omitted. Only those portions of the program necessary to illustrate the desired point will be shown. Sometimes indentation alone will indicate nesting.

```
PROGRAM A;
PROCEDURE B;
PROCEDURE C;
    PROCEDURE D;
        PROCEDURE E;
            PROCEDURE F;
    PROCEDURE G;
PROCEDURE H;
BEGIN
  ...
END
```

This example shows a program (A) with three procedures (B, C, and H). Procedure C contains two procedures (D and G). Procedure D contains a procedure (E) which contains another procedure (F).

Program A (i.e., the statements in the main program) can use only those variables declared in block A (i.e., global variables). Procedure B can use its local variables and the variables declared in A. The same is true for procedures C and H. Procedure D can use its local variables and variables declared in C or A. The same is true for procedure G. Procedure E can use its local variables and variables declared in D, C, or A. Finally, procedure F can use its local variables and variables declared in E, D, C, or A.

As mentioned previously, the scope of an identifier, also called its domain of accessibility, is the block within which it is declared. Since these blocks may contain procedures and functions:

- Variables declared in A are available to A, B, C, D, E, F, G, and H.
- Variables declared in B are available to B alone.
- Variables declared in C are available to C, D, E, F, and G.
- Variables declared in D are available to D, E, and F.
- Variables declared in E are available to E and F.
- Variables declared in F are available to F alone.
- Variables declared in G are available to G alone.
- Variables declared in H are available to H alone.

4. A block will use the most local identifier. If more than one variable is declared with the same name, the most local identifier will take precedence. Let's assume that we are executing procedure E in the previous example and a reference is made to the variable X. If there is a variable named X declared locally to procedure E, then this is the variable that will be used, even if there are other variables named X in the program. If there is no X declared in procedure E, then procedure D will be examined. If an X is declared in procedure D, it will be used; otherwise procedure C will be examined. If an X is declared in procedure C, it will be used; otherwise the global variables of A will be examined. If a global X is declared in program A, it will be used; otherwise an error message will appear indicating an undeclared identifier.

5. In a previous lesson it was stated that a local variable exists while its procedure or function is executing. This existence should not be confused with its scope.

```
PROGRAM X;
VAR
  A : integer;
PROCEDURE Y;
  BEGIN
    A := A + 2
  END;

PROCEDURE Z;
VAR
  A : integer;
BEGIN
  ...
  Y
  ...
END;

BEGIN
  ...
  Z
  ...
END. (* X *)
```

In this program, the main program invokes procedure Z, which in turn invokes procedure Y. Procedure Y increments A by 2. But which A gets incremented? Based on our previous discussion, it should be clear that the global A gets incremented. The A declared in procedure Z is only available to procedure Z, even though it still exists while procedure Y is executing, since procedure Z has to wait for procedure Y to be completed before it can continue. Also note that an undeclared - identifier error message would have been generated if there was no global variable named A, even though procedure Z has an A declared.

6. We have seen that Pascal allows procedures and functions to use nonlocal identifiers (i.e., identifiers that are not declared as parameters or local variables). Although this is legal, it is not good programming style. Nonlocal identifiers should always be passed as parameters. The use of nonlocal identifiers causes side effects. These side effects should be avoided. It is also a good practice to avoid using the same identifier name more than once whenever practical. The computer will never get confused, but the programmer might.

Summary:

1. What is a global identifier?
2. What is a local identifier?
3. What is a block?
4. What is the scope of an identifier?
5. What is a side effect?
6. How can side effects be avoided?
7. What is the difference between the domain of accessibility and the existence of an identifer.

Exercises:

1. What will be the output of this program?

```
PROGRAM Test;
VAR
  A, B, C, D : integer;

PROCEDURE N (A : integer; VAR B, C : integer);
BEGIN
  A := A + 1; B := B + 2; C := C + 3; D := D + 4;
  writeln (A, B, C, D)
END;

BEGIN
  A := 2; B := 3; C := 4;
  D := A + B;
  writeln (A, B, C, D);
  N (D, C, A);
  writeln (A, B, C, D);
END. (* Test *)
```

2. What is the scope of the identifiers in A, B, C, D, E, and F below?

```
PROGRAM A;
PROCEDURE B;
  PROCEDURE C;
  PROCEDURE D;
    PROCEDURE E;
PROCEDURE F;
BEGIN
  ...
END. (* A *)
```

3. Write a function that will take on the value of the largest of four integer parameters. Write this function so that it contains a function that will take on the value of the larger of two integer parameters.

4. Write a procedure that will reduce fractions that have their numerator and denominator declared globally. Your procedure should replace the unreduced values by their reduced values. Write the procedure so that it contains a function to calculate the greatest common factor of two integers.

Topic 28

FORWARD REFERENCES

Performance Objectives: Students will be able to:

- state the purpose of forward references
- describe how forward references are implemented
- explain how functions and procedures can be used as parameters

Vocabulary: forward reference, FORWARD, indirect recursion

Motivation:

- Answer the questions below based on the following program structure:

```
PROGRAM A;
PROCEDURE B;
  BEGIN
    ...
  END;
PROCEDURE C;
  PROCEDURE D;
    BEGIN
      ...
    END;
  BEGIN
    ...
  END;
BEGIN
  ...
END. (* A *)
```

(a) Which block(s) (A, B, C, D) can invoke procedure B?
(b) Which block(s) (A, B, C, D) can invoke procedure C?
(c) Which block(s) (A, B, C, D) can invoke procedure D?

- These are the solutions:

(a) A, B, C, D
(b) A, C, D
(c) C, D

Aim: How can forward references be used with procedures?

138

Development:

1. In the motivation problem, procedure C can invoke procedure B, but procedure B cannot invoke procedure C. Recall that a procedure can invoke any of its local procedures, itself, and any previously declared procedures. If procedure X needs procedure Y, we can do one of two things. If only procedure X needs procedure Y, then Y should be declared locally as part of X. On the other hand, if procedure Y is also needed by the main program or other procedures, then procedure Y should be declared globally before procedure X and any other procedures that need it.

   A problem arises if procedure X needs procedure Y and procedure Y needs procedure X. Which procedure do we declare first? The previous rules for procedure availability do not allow us to resolve this dilemma. But Pascal has something called a forward reference. This is a statement that notifies the complier that a certain procedure is to be made available now but will not be declared until further on in the program.

2. To make a forward reference, write the statement

   ```
   FORWARD;
   ```

   following the usual procedure or function heading (including any parameters). Then declare the procedure or function later on in the program in the usual manner, except that the parameters are not repeated. For example:

   ```
   PROCEDURE Y (A, B : integer; VAR C : char);
     FORWARD;

   PROCEDURE X (D : integer);
   BEGIN
     ...
   END;

   PROCEDURE Y;
   BEGIN
     ...
   END;
   ```

   Now procedure X can invoke procedure Y and procedure Y can invoke procedure X. If procedure X invokes procedure Y and procedure Y invokes procedure X, we have a form of recursion called indirect recursion (to be discussed in lesson 34).

3. Another feature of Pascal is the ability to pass procedures and functions as parameters to other procedures or functions. Each such parameter must be preceded by the key word PROCEDURE or FUNCTION in the parameter list. The parameters of these parameters must be value parameters. For example:

```
FUNCTION Max (FUNCTION F : real; A, B, delta : real) : real;
VAR
  large : real;
BEGIN
  large := F(A);
  A := A + delta;
  WHILE A<= B DO
    BEGIN
      IF F(A) >large THEN
        large := F(A);
      A := A + delta
    END;
  Max := large
END; (* Max *)
```

This function will return the largest value of a real function in the interval from A to B in increments of delta. For expediency, tests for B>= A and delta > 0 have been omitted. Note that the parameter list for the function parameter F is not listed in the parameter list for Max. Some implementations of Pascal may require this parameter list, but it is usually not required. The use of F in the function Max dictates that F is to have one real parameter. Therefore, a call to Max must provide an actual parameter which is a real valued function with one real value parameter along with three real actual parameters. For example:

```
Max (sin, 0, 1, 0.1)
```

will return the largest value of sin from 0 to 1 radian by testing increments of .1 radian.

These procedure and function parameters are not used often but can be very useful in mathematical programs.

Summary:

1. Why are forward references needed?
2. How do we make a forward reference?
3. How do we pass procedures and functions as parameters?

Exercises:

1. Assume the following:

```
PROCEDURE A (A, B : integer)
FUNCTION B (VAR A : integer; C : integer) : char
FUNCTION C (X : real) : integer
PROCEDURE D (X, Y : char)
```

Write the skeleton declarations (i.e., just the headings) that declare A and B with forward references, then C and D, and finally A and B.

2. Write a real valued function which has two function parameters (real valued functions with one real parameter) and one real value parameter. The function should take on the larger of F(X) or G(X) where F and G are the two function parameters and X is the real parameter.

3. Write a boolean valued function which has two function parameters (real valued functions with one real parameter) and one real parameter. The function should take on the value of TRUE if F(G(X)) = G(F(X)), and FALSE otherwise. F, G, and X are the parameters.

Topic 29

TOP-DOWN DESIGN

Performance Objectives: Students will be able to:

- state the advantages of top-down design
- contrast the top-down and bottom-up approaches to problem solving
- define program modularization and discuss its benefits

Vocabulary: top-down design, bottom-up design, modularization, portability

Motivation:

- Have students describe the technique they would follow in writing an autobiographical composition.

- One student may decide to separate her life into the stages of infancy, prepuberty, and adolescence and then subdivide each stage into categories such as positive influences, negatives influences, and human-interest activities. Each of these categories can be further subdivided into actual events that occurred and their interrelationships. Another student may decide to list specific events which occurred in his life and then try to interweave them into more cohesive units. This methodology is the reverse of that applied by the first student.

  Both techniques are useful ways to approach the problem at hand. They are used extensively in project design, especially by programmer-analysts.

- The first technique refines a general problem into broad categories and defers details until a later stage. It is commonly called top-down design. The second technique reverses the process. It is called bottom-up design.

Aim: What are the differences between the top-down and bottom-up approaches to problem solving?

Development:

1. As we proceed through the course the problems that we encounter will become increasingly more difficult, and they will not be amenable to easy solution by the somewhat haphazard methods used previously. Although there are several systematic methods for solving problems, the hierarchal structure of most computer programs makes the top-down design approach especially suitable. In the top-down method a general solution to a computer problem is stated and successively refined into subproblems until each subproblem has a simple direct solution in the programming language. The most straightforward way to accomplish this refinement is through the use of procedures. For example, the general solution to a programming problem might use these four procedures:

- Inputvalues
- Performcalculations
- Findstatistics
- Printresults

Each of these could then be further refined into other statements and procedures that give a more detailed solution to the problem. These procedures help the programmer retain an overall perspective on how the solution to the original problem was reached. The last step in the refinement is the actual coding of the solution in the programming language. For example, the module Inputvalues might contain these specific subtasks:

- get input data
- check for bad data

The subtask "check for bad data" would contain these subtasks:

- test for bad data
- give appropriate message to user

The top-down method allows the programmer to see the overall solution to the program together with its complex relationships. Programs must be tested and may interface with other programs. Stepwise refinement is an essential feature of the top-down design approach. Top-down design explores the algorithmic solution at all levels.

2. A major advantage of top-down design is that it can help the programmer realize that an apparently plausible approach is incorrect. Since the cost of making cosmetic changes in a poorly designed program increases exponentially in the implementation stage, it is preferable at the beginning of the developmental process to backtrack and redesign or simply throw out a badly designed solution.

   Another important benefit of top-down design is the identification of separate and distinct modules through the process of stepwise refinement. The modularization of a program through the use of relatively independent subprograms (procedures, functions) makes the program easier to write, understand, test, debug, and maintain. Because global identifiers cause side effects and spoil modularity, parameters should be used in subprograms as often as possible. In addition, modules from previous programs may be advantageously reused to simplify the writing of current programs.

3. In contrast to the top-down method, the bottom-up method proceeds from lower-level subprograms to those at the higher level. This method is often used when the appropriate approach to the solution of the problem is unclear. In this situation a programmer may decide to experiment with operations on specific data structures to test their feasibility and obtain a better picture of implementation difficulties. These lower-level procedures can then be used to write procedures that are higher in the hierarchy.

4. The following is an example of top-down design used to outline the steps in writing a bank program that processes customer transactions.

The input to the program consists of several account numbers and opening balances, each followed by transactions which pertain to that particular account. The transactions can be of two types: 'D' followed by amount deposited or 'W' followed by amount withdrawn; any other character indicates the end of the transactions for that account. At the end of the processing of transactions for each account a table should be printed indicating the final balance, as well as the number and value of deposits, withdrawals and overcharges. Withdrawals that make the balance negative should <u>not</u> be processed, but a $10 fee should be added to the overcharges. The processing is complete when a sentinel is encountered.

This would be a typical run:

```
Enter account number (0 to end):  3482
Opening balance:  $3400
Enter transactions--W or D followed by amount:
W    200
D    316.80
W    5000
Overdrawn!  (not processed) special charge:  $ 10.00
D    24.05
E
Final balance (# 3482):  $3530.85

Deposits:  2            Amount of deposits:  $340.85
Withdrawals:  1         Amount of withdrawals:  $200.00
Overcharges:  1         Amount of overcharges:  $ 10.00

...
Enter account number (0 to end):  0
```

- In the first design the programmer lists in order the major steps that will be needed to solve the problem:

<u>Design 1</u>

- Input account ID numbers and opening balances.
- Input transactions of deposits and withdrawals.
- Process transactions.
- Print account summaries.

- At this point, the problem in design 1 can be broken down into more precisely stated subproblems through stepwise refinement: For example:

<u>Design 2</u>

- Read first account ID number and opening balance.
- Initialize variables.
- Enter first transaction code for account. Use W for withdrawals, D for deposits, and any other letter for end of transaction.

- Add deposits.
- Subtract withdrawals and service charges.
- Print account summary for specific account ID number.

. While design 2 is more detailed than design 1, it is still too abstract for the programmer, and it needs further refinement:
<u>Design 3</u>

```
- Read first account ID number.

  WHILE there are more accounts to process DO
    Read opening balance
    Initialize variables
    Enter first transaction code for account (W or D)
    WHILE there are more transactions DO
      Read transaction amount
      CASE
        D : Add deposit to amount of deposits
            increment number of deposits by one
            add deposit to balance
        W : Subtract withdrawal from balance
            IF balance is negative THEN
              withdrawal is added back to balance
              subtract $ 10 from balance
              add $ 10 to overcharges
              increment number of overcharges by one
            ELSE
              subtract withdrawal from amount of withdrawals
              increment number of withdrawals by one
      Read next transaction code
    Print summary of account transactions
    Read next account id number
```

. At this point, the programer is ready to write a program using the top-down method:

```
PROGRAM Banktransactions (input, output);
(* Bank transactions are processed *)
(* for a set of separate accounts *)
CONSTANT
  acctsentinel = 0;
VAR
  acctid : integer;
  balance : real;

PROCEDURE Processtransactions (acctid : integer; balance : real);
CONST
  overcharge = 10.0;
VAR
  numdeposits, numwithdrawals, numovercharges : integer;
  amtdeposits, amtwithdrawals, amtovercharges : real;
  transamt : real;
  transcode : char;
```

```
PROCEDURE Printsummary;
BEGIN
  writeln;
  writeln ('Final balance (#', acctid : 6, '):  $', balance : 6 :
  2);
  writeln;
  writeln('Deposits:', numdeposits : 3, 'Amount of deposits:  $' :32,
  amtdeposits : 6 : 2);
  writeln('Withdrawals:', numwithdrawals : 3, 'Amount of
  withdrawals:  $':29, amtwithdrawals : 6 : 2);
  writeln('Overcharges:', numovercharges : 3, 'Amount of
  overcharges:  $':29, amtovercharges : 6 : 2);
  writeln
END; (* Printsummary *)


BEGIN (* Process transactions *)
  (* initialize variables *)
  numdeposits := 0; numwithdrawals := 0; numovercharges := 0;
  amtdeposits := 0.0; amtwithdrawal := 0.0; amtovercharges := 0.0;
  (* enter first transaction *)
  writeln ('Enter transaction--W or D followed by amount:');
  read (transcode);
  WHILE (transcode = 'D') OR (transcode = 'W') DO
    BEGIN
      readln(transamt);
      CASE transcode OF
        'D' : BEGIN (* deposit *)
                amtdeposits := amtdeposits + transamt;
                numdeposits := numdeposits + 1;
                balance := balance + transamt
              END; (* CASE 'D' *)
        'W' : BEGIN (* withdrawal *)
                balance := balance - transamt;
                IF balance < 0 THEN
                  BEGIN
                    writeln ('Overdrawn!  (not processed) special
                    charge:  $', overcharge : 6 : 2);
                    balance := balance + transamt - overcharge;
                    amtovercharges := amtovercharges + overcharge;
                    numovercharges := numovercharges + 1
                  END (* THEN *)
                ELSE
                  BEGIN
                    amtwithdrawals := amtwithdrawals + transamt;
                    numwithdrawals := numwithdrawals + 1
                  END (* ELSE *)
            END (* CASE 'W' *)
      END; (* CASE *)
      read (transcode)
    END; (* WHILE *)
  Printsummary
END; (* Processtransactions *)
```

```
BEGIN (* Banktransactions *)
  writeln ('Enter account number (0 to end):');
  readln (acctid);
  WHILE acctid<>acctsentinel DO
    BEGIN
      write ('Opening balance:');
      readln (balance);
      Processtransactions (acctid, balance);
      write ('Enter account number (0 to end):');
      readln (acctid)
    END (* WHILE *)

END. (* Banktransactions *)
```

Summary:

1. Describe how top-down design might be applied to a problem.

2. What are the advantages of programming in modules?

Exercises:

1. Use top-down design and stepwise refinement to analyze the following problems:

   (a) planning a birthday party for 20 people
   (b) going out for dinner
   (c) planning a trip to Europe

2. Look up the following terms in the text Programming in Pascal* by Peter Grogono: "divide & conquer," make finite progress, "use CASE analysis."

---

*Grogano, Peter. Programming in Pascal (Revised Edition.)
(Reading, Mass: Addison-Wesley Publishing Co., 1984.)

Topic 30
PROGRAMMING METHODOLOGY

Performance Objectives: Students will be able to:

- list the essential elements of good programming methodology
- discuss the concepts of specification, design, coding, correctness, and documentation

Vocabulary: top-down design, specification, documentation, internal documentation, external documentation

Motivation:

- Outline the steps you would take to solve a computer problem. (Elicit responses.)

- A typical outline might include:

  - understanding the problem
  - developing an algorithm to solve the problem
  - refining the algorithm
  - coding the algorithm in Pascal

- Although such an outline might be suitable for writing simple programs, it has limitations for producing quality programs, whether for the student's personal use or for use in a business setting. A more formal set of techniques is needed for developing software systems that satisfy the needs of the user over an extended period of time. Programming methodology addresses this concern by concentrating on those aspects of program development that make software highly reliable and easy to maintain and modify.

Aim: What are the essentials of good programming methodology?

Development:

1. There are five main aspects of good programming methodology that will be analyzed: specification, design, coding, correctness, and documentation.

   - Specification: Wouldn't it be annoying to work very hard on a term project and then be told by the teacher, "That's not what I wanted" or "Your format is all wrong" or "Where are the footnotes and bibliography?" The same type of disastrous situation multiplied manyfold occurs frequently between a programmer and the user for whom the program is being written. To avoid such difficulties, the programmer needs a clear specification of the problem to be solved. This specification should contain the logical requirements of the program, as well as any special needs of the user, without dictating a physical implementation. Since the cost of changing the functional characteristics of a program can grow exponentially the closer one gets to the finished product, a clear specification agreed upon by the user and designer is essential.

Most large companies have systems analysts whose job it is to write lucid specifications and serve as the interface between the user and systems designer or programmer. In a typical commercial setting, the user knows nothing about programming, and the programmer knows little about a specific application. It is the systems analyst who attempts to understand the user's needs and communicate these needs to a designer in a specification document.

A typical specification for a dating program might state that the user should have the option of selecting 20 to 30 questions for a dating application form. Some questions may be user-defined, while others may come from a data base of 50 preprogrammed questions. Although each match between applicants should have a default score of one, the user can increase point scores up to three for any question. Another option should preclude any matches for dissimilar answers to selected questions. In addition, each respondent who fills out a dating form should receive a printout of the five best matches, together with point scores.

- Design: The design stage involves converting the specifications into a carefully planned design that illustrates how the program will accomplish its task. We previously discussed how the emphasis in top-down design and stepwise refinement leads to the development of clearly defined modules that are convenient to program and easy to maintain. (See Topic 29 for a more complete discussion of design considerations.)

- Coding: Coding is the process of transforming a design into a program that can be executed by the computer. Although coding is often considered to be the complete programming process, it has been estimated that only one-sixth of the time needed to develop large software systems is devoted to coding, while the remaining five-sixths is devoted to planning and testing. A programmer should always use proper indentation, meaningful identifiers, and logical constructs to make the program more readable. Segments that take the least time to code are not always the best. They may make the program more difficult for someone other than the original author to understand, maintain, and modify. In addition, coding time may be relatively insignificant when storage space and running time are the primary considerations for a program that has to be executed repeatedly.

- Correctness: Programs are correct when they meet specification requirements. Testing provides a means to measure program correctness; the more testing that's done, the more we can feel confident that the program is correct. As impractical as it is to provide exhaustive testing for a program that has thousands of possible input values, we should nevertheless attempt to provide a sufficient number of test cases to insure program viability. (Further concepts related to correctness and testing will be discussed in the next topic.)

- Documentation: An often neglected but crucial aspect of programming methodology is good program documentation. Just as we all get bored and annoyed when someone communicates ineffectively, so should we be irritated when a programmer doesn't indicate what task a program is supposed to perform and how it accomplishes that task. Documentation serves many individuals: It refreshes the memory of the person who wrote the program; it explains design and functional characteristics to someone other than the programmer who might have to debug and update the program at some future time; and it tells the user about the operation of the program and program options. In many companies, over 50% of programming time is spent on program maintenance because of poor documentation in previously written programs.

What should be included in internal documentation (comments within the program)?

- a statement of the problem that the program solves
- the purpose and logical structure of each module or collection of statements
- a short description of all identifiers
- special program assumptions (e.g., input values, operating instructions, file system)

External documentation can include such information as user options, operating environment, run-time efficiency, program listings, and illustrated test cases.

2. Professionals in the computer field have shown that good programming methodology produces quality software. Students should be encouraged to use these techniques in writing programs.

Summary:

1. What do the following programming methodology terms refer to: specification, design, coding, correctness, documentation?

2. What types of documentation should be included in a program?

Exercises:

1. What would be the specifications for a program that plays tic-tac-toe? List as many specifications as possible. Assume that the programmer does not know how to play the game.

2. Write specifications for a program whose object is to add two given fractions and print out their sum as a mixed number.

3. Write specifications for a program that plays the card game "twenty-one."

Topic 31
PROGRAM CORRECTNESS

Performance Objectives: Students will be able to:

- differentiate between several levels of program correctness
- discuss the process of program testing
- describe various types of errors and diagnostic techniques

Vocabulary: driver program, stub, bug, debugging, tracing, invariants, top-down testing, bottom-up testing, off-by-one errors

Motivation:

- What do you think it means for a program to be correct? (Elicit responses)

Aim: What is meant by program correctness?

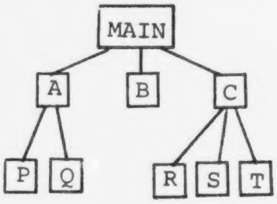
Development:

1. The programmer and user often attach different meanings to program correctness. Several interpretations of correctness are listed below in increasing levels of rigor:

   (a) There are no syntax errors during compilation.
   (b) There are no detectable errors during execution.
   (c) The program works for some test data.
   (d) The program works for a well chosen set of typical data.
   (e) The program works for difficult sets of test data.
   (f) The program works for all possible sets of test data and typical incorrect input.
   (g) The program works for all possible input.

   Note: What is an appropriate level of correctness for student programs? Students should be made aware that criteria (a) through (c) are insufficient for determining program correctness. Although the teacher may specify more detailed error checking for projects, the cluttering of regular program assignments with numerous input checks can be very tedious. Higher levels of correctness are needed, however, in commercial applications. In general, a program is correct if it meets all of its specifications.

2. To show that programs are correct, we must do testing. What are techniques that a programmer can use to test programs? Suppose that a program has the design illustrated in this figure, where the letters indicate subprograms:



Recall that if we used top-down design, the major subprograms (A-C) would have been designed first and then refined stepwise into the lower order modules (P-T); bottom-up design would have proceeded in the reverse direction. Similarly, there are two major techniques for testing programs: top-down testing and bottom-up testing.

3. In bottom-up testing we start by testing each of the lower-level modules. To do this we need a short program that inputs values into the subprogram and then prints out the results. This is called a driver program. A driver program that tests procedure P might take this form:

```
PROGRAM TestP (input, output);
VAR
  X, Y : integer;

PROCEDURE P (VAR A, B : integer);
.
.
.
BEGIN (* TestP *)
  X := 3;
  Y := 5;
  P (X, Y);
  writeln (X, Y)
END. (* TestP *)
```

4. We can now use an incremental approach by testing subprogram Q, then testing the combination of P and Q (i.e., A), until all modules have been tested together. The extra effort expended in writing driver programs pays dividends by reducing the overall testing time. Without this bottom-up method, errors in lower-level modules are much harder to detect when the whole program is run.

5. Top-down testing requires that the top-level modules be coded and tested first. In our figure, A, B, and C would be tested before modules P-T are implemented. How is this possible? At the time that Main is tested, modules A, B, and C must be specified and their interaction with Main known. However, A, B, and C may not yet be coded or may simply exist in skeletal form or as dummy modules. Therefore, in order to test the main program we write procedures that simply return a specific value. These values are not necessarily correct, but they do provide some value so that program execution may continue. These programs, which are a rough implementation of the actual program, are called stub programs. Although the program's modules are incomplete, they help demonstrate that the program design is correct. For example, in writing a stub program for procedure C, which is supposed to calculate a tax rate on various incomes, we may simply fix the tax rate at 20%.

6. Which is preferable--top-down or bottom-up testing? It is best for beginners to design top down and test bottom up. The use of drivers to test modules is comparatively simple and helps the programmer concentrate on one procedure at a time. Top-down testing is usually preferable for large projects, since it allows testing to go on while

the program is being written. Several other advantages of top-down testing are that it tests the most important things first, it allows easier integration of program modules, and it allows users to see a preliminary version of the program.

7. After the program has been coded, the programmer needs to test it for errors. The skillful programmer will generate a sufficiently difficult set of test cases to insure program correctness. The programmer should try to make the program fail by selecting extreme and special values. For example, in a program that works for values between 100 and 150, the programmer should use test data such as 85, 99, 100, 101, 128, 149, 150, and 151. Test data at interval boundaries is important because off-by-one errors (OBOEs) are common. An off-by-one error is an error that occurs one unit away from a boundary. It is usually caused by the programmer's using the wrong relational operator as a test (e.g. $<$ rather than $<=$). Errors that occur while testing are called bugs, and it is the programmer's job to fix these errors by debugging the program.

8. What are the different types of errors that may occur during testing?

   - errors in language syntax
   - errors in understanding the programming language (e.g., confusing value and variable parameters)
   - errors in program design (e.g., improperly invoking subprograms)
   - errors due to carelessness (e.g., leaving out a line of a program)

9. What are some debugging aids? The compiler not only pinpoints certain syntactic errors, but often has special built-in directives that enable the programmer to locate additional bugs, such as improper nesting levels. A frequently used technique that enables a programmer to check the logical structure of the program is called tracing. Tracing is the process of executing the program one instruction at a time. It allows us to stop the execution at a given point in the program and examine the values of the variable at that point. By noticing when variables are assigned incorrect values during execution, we can often determine the cause of the error. The most popular method of locating errors is to print intermediate results by inserting writeln statements in appropriate places throughout the program. It is also useful to echo print input data to see if the proper data is being read in.

10. As we have discussed, program testing involves locating errors that have to be debugged. Although exhaustive testing can increase confidence that the program works, it cannot guarantee it. Because of this difficulty, several computer scientists are working on methods to verify program correctness using proofs similar to those used in mathematics. These proofs frequently use the idea of invariants. Invariants of a loop are expressions whose truth value remains unaltered during execution of the loop. For example:

```
(* An array A with components to N has been declared *)
(* A[I] denotes the Ith element of array A *)
N := 50;
I := 1;
total := A[1];
WHILE I < N DO
  BEGIN
    I := I + 1;
    total := total + A[I]
  END;
```

Total contains the sum of values in A[1]..A[I] where 1<= I<= N. (Note: A[1]..A[I] means all the values from A[1] to A[I] inclusively.) (Further examples will be covered when studying the topic of arrays. References in the appendix provide a more comprehensive treatment. See, for example, Grogono and Conway, et al.)

Summary:

1. When is a program correct?
2. What is top-down and bottom-up testing?
3. What is a driver program?
4. What is a stub program?
5. What is an off-by-one error?
6. What is debugging?
7. What is a trace?
8. What is an invariant?

Exercises:

1. What is wrong with the following program segment, which is supposed to sum numbers from 1 to 100?

```
I := 0;
WHILE I<= 100 DO
  BEGIN
    I := I + 1;
    sum := sum + I
  END;
```

2. Assume that we have a function called Gcf that returns the greatest common factor of two integers. The function heading for Gcf is:

```
FUNCTION Gcf (X, Y : integer) : integer;
```

Write a driver program that will allow you to input various pairs of integers to test the function. The driver program should terminate when the values 0, 0 are input.

3. We want to test a stub program which has a function to calculate tax rates, but the function has not yet been written. The function heading is:

```
FUNCTION Taxrate (salary : real) : real;
```

To test this stub program, write a dummy function which returns a tax rate of 20%.

Topic 32

RECURSIVE FUNCTIONS: PART 1

Performance Objectives: Students will be able to:

- define a recursive function
- explain how recursion works for finding factorials
- discuss when recursion can be used in problem solving

Vocabulary: recursion, recursive function

Motivation:

- How does the function Power below differ from other functions that we have previously encountered? (Elicit responses.)

```
PROGRAM Calculation (output);
FUNCTION Power (N : integer) : integer;
  BEGIN
    IF N = 0 THEN
      Power := 1
    ELSE
      Power := 2 * Power (N - 1)
  END;

BEGIN (* Calculation *)
  writeln (Power (4))
END. (* Calculation *)
```

- In the function Power, the name of the function not only appears on the left side of an assignment statement, but also on the right side. The function calls itself from within the function. Since the scope of a function name includes its own block, the function name may be used within the function. A function that calls itself is called a recursive function. (Similarly, there are recursive procedures).

Aim: What is a recursive function?

Development:

1. The term recursion is used frequently in mathematics for formulas that are defined in terms of themselves. This is a typical recursive definition used for factorals of nonnegative integral values:

$$N! = \begin{cases} 1 \quad \text{IF} \quad N = 0 \\ N * (N - 1)! \quad \text{IF } N >= 1 \end{cases}$$

If we let Fact (N) = N!, the recursive relationship can be rewritten as:

```
Fact (N) = 1  IF N = 0
           N * Fact (N - 1)  IF N>= 1
```

2. 5! can now be evaluated using this recursive definition:

```
Fact (5) = 5 * Fact (4)
         = 5 * 4 * Fact (3)
         = 5 * 4 * 3 * Fact (2)
         = 5 * 4 * 3 * 2 * Fact (1)
         = 5 * 4 * 3 * 2 * 1 * Fact (0)
         = 5 * 4 * 3 * 2 * 1 * 1        = 120
```

3. How can we write a Pascal recursive function that finds factorials? (Elicit responses.)

   Answer:

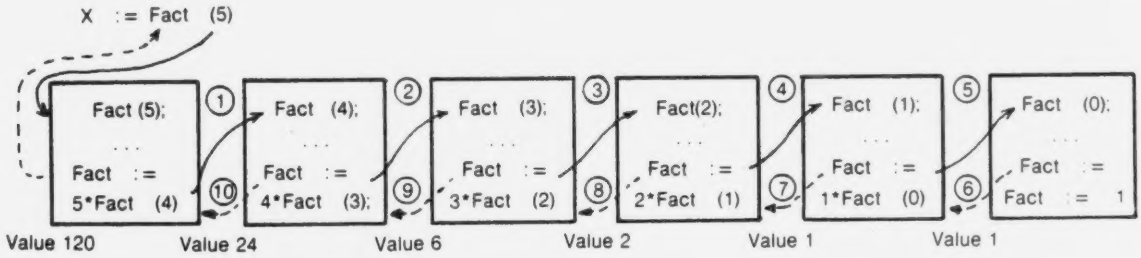
```
FUNCTION Fact (N : integer) : integer;
(* For nonnegative integral values of N *)
BEGIN
  IF N = O THEN
    Fact := 1
  ELSE
    Fact := N * Fact (N - 1)
End; ( * Fact * )
```

   This function can be called by a statement such as this in the main program:

```
  X := Fact (5)
```

   The value returned to X would be 120.

4. Let's examine the behavior of this function call:


X := Fact (5)
Fact (5);
...
Fact :=
5*Fact (4)
①
Fact (4);
...
Fact :=
4*Fact (3);
②
Fact (3);
...
Fact :=
3*Fact (2)
③
Fact(2);
...
Fact :=
2*Fact (1)
④
Fact (1);
...
Fact :=
1*Fact (0)
⑤
Fact (0);
...
Fact :=
Fact := 1
⑥
⑦
⑧
⑨
⑩
Value 120
Value 24
Value 6
Value 2
Value 1
Value 1

Numbers one to five represent five recursive calls. For each call, the values of all local variables are saved in the computer's memory until a value has been returned by the invoked function and the calling function has been exited. The last recursive call occurs when the parameter has a value of zero. As functional values are returned to the calling programs (six to ten), the levels of recursion decrease and the value of 120 is eventually returned to the original (nonrecursive) function call.

5. What types of problems can be solved by recursion? Recursion can be used when the solution can easily be given for certain simple cases and when there are specific rules for proceeding from a given state to a new state that is either a simple case or that eventually leads to a simple case. These simple cases are usually the smallest (or largest) values for which the problem is defined. In the factorial example, Fact (0) = 1 is the simple case and Fact (N - 1) is closer to the simple case than Fact (N).

6. Let's look more closely at the recursive calls in the Power function we examined earlier. The function Power (N) recursively calculates 2 to the Nth power:

```
Power (N) = 1                     IF N = 0
Power (N) = 2 * Power (N - 1)  IF N >= 1
```

```
Power (4) = 2 * Power (3)
          = 2 * 2 * Power (2)
          = 2 * 2 * 2 * Power (1)
          = 2 * 2 * 2 * 2 * Power (0)
          = 2 * 2 * 2 * 2 * 1 = 2^4   = 16
```

Recursion can be successively applied to this problem because there is a simple case and a rule that eventually leads to a simple case.

Summary:

1. What is a recursive function?

2. What types of problems can be solved using recursion?

Exercises:

1. Evaluate 4! using the recursive function written in class.
2. Evaluate 2^5 using the recursive function written in class.
3. Write a recursive function that multiplies two positive numbers, A and B, using the definition:

$$A * B = \begin{cases} A & \text{IF } B = 1 \\ A * (B - 1) + A & \text{IF } B > 1 \end{cases}$$

4. Find 7 * 3 using the recursive function from Exercise 3.

Topic 33
RECURSIVE FUNCTIONS: PART 2

Performance Objectives: Students will be able to:

- explain how recursion works for finding Fibonacci numbers
- discuss the advantages and disadvantages of recursion
- write simple recursive functions

Vocabulary: stack overflow error, Fibonacci sequence

Motivation:

- Using the following sequence, determine the next three items: 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, __, __, __. (Elicit responses.)

- The next three terms are 89, 134, and 223. This sequence is the first in an interesting series of numbers called the Fibonacci sequence. This sequence has been examined extensively by mathematicians for the past several hundred years. While the first two terms are both 1, each number, starting with the third term, can be found by taking the sum of the two previous terms. The Fibonacci sequence can therefore be defined recursively as:

$$\text{Fib }(N) = \begin{cases} 1 & \text{IF } N = 1 \text{ OR } 2 \\ \text{Fib }(N - 1) + \text{Fib }(N - 2) & \text{IF } N > 2 \end{cases}$$

  Where Fib (N) denotes the Nth Fibonacci number.

- A Pascal function that implements this recursive definition to help us find the Nth term of the Fibonacci sequence is:

```
FUNCTION Fib (N : integer) : integer;
  BEGIN
    IF (N = 1) OR (N = 2) THEN
      Fib := 1
    ELSE
      Fib := Fib (N - 1) + Fib (N - 2)
  END;
```

Aim: Why use recursive functions and procedures?

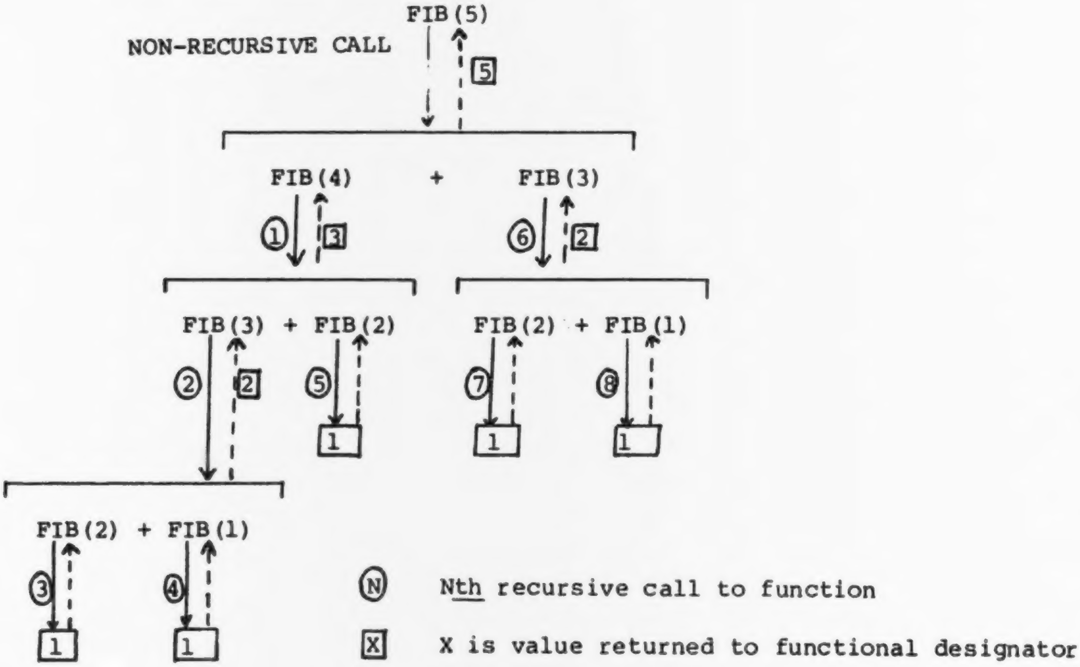
Development:

1. Once a solution to a problem is formulated recursively, it is relatively easy to write a Pascal program that implements the solution.

2. How many recursive calls would occur using this statement in the main program? (Elicit responses.)

```
writeln ('The fifth Fibonacci number is ', Fib (5));
```

There will be eight recursions. The value returned to Fib (5) would be 5, the fifth term of the sequence:


FIB(5)
NON-RECURSIVE CALL
5
FIB(4) + FIB(3)
① 3
⑥ 2
FIB(3) + FIB(2)
FIB(2) + FIB(1)
② 2
⑤
⑦
⑧
1
1
1
FIB(2) + FIB(1)
③
④
1
1
Ⓝ Nth recursive call to function
☒ X is value returned to functional designator

3. What are the advantages and disadvantages of using recursion? One disadvantage is that recursion uses a large amount of computer memory. Every time a function is called recursively, a new set of local variables is created. In addition, the address of the function designator is saved so that the computer knows where to continue program execution after a value is returned by the function. When a long sequence of recursive calls uses up all of the available memory set aside for local variables and return addresses, a stack overflow error occurs and the computer stops execution. Another disadvantage of recursion is the excessive amount of time involved in allocating and deallocating storage. However, recursive functions are frequently used in programs because they are easier to write and easier to understand. Thinking recursively can also lead to simpler nonrecursive solutions.

4. What would be output by the procedure if we entered the sentence, "The cat meows."? (Elicit responses.)

```
PROGRAM Guesswhat (input, output);

PROCEDURE Rev;
VAR
  ch : char;
BEGIN
  read (ch); (* UCSD: read (keyboard, ch); *)
  IF NOT eoln THEN
    Rev;
  write (ch)
END; (* Rev *)

BEGIN
  writeln ('Enter a sentence.');
  Rev;
  writeln
END.  (* Guesswhat *)
```

A careful trace of the recursive calls shows that the input would be reversed and the computer would print ".swoem tac ehT"

Summary:

1. What are the advantages of using recursion?

2. What are the disadvantages of using recursion?

Exercises:

1. Write a nonrecursive version of the function Fib from this lesson.

2. Write a recursive function that calculates $X^N/N!$

3. Analyze the following procedure. What would be output by the procedure call Bin (25)?

```
PROCEDURE Bin (N : integer);
BEGIN
  IF N> 0 THEN
    BEGIN
      Bin (N DIV 2);
      write (N MOD 2)
    END
END; (* Bin *)
```

<u>Solution:</u>

This procedure changes N to binary form. The output for the call Bin (25) would be 11001. This procedure is similar to the Rev procedure discussed earlier in the topic since the remainders are written in reverse order. We repeat division by 2 until the quotient is zero, and we write the remainders in reverse order:

```
2 25
  12  1
   6  0
   3  0                 11001
   1  1
   0  1
```

Topic 34

INDIRECT RECURSION

Performance Objectives: Students will be able to:

- define indirect recursion
- explain how forward references are used with indirect recursion
- describe how a recursive function stacks data items

Vocabulary: indirect (mutual) recursion, stack, forward reference

Motivation:

- Suppose F and G are two functions defined for positive integral values of X. Are functions F and G recursive? (Elicit responses.)

$$F(X) = \begin{cases} G(2 * X) + 4 & \text{If } X \leq 20 \\ F(X \text{ DIV } 3) - 1 & \text{If } X > 20 \end{cases}$$

$$G(X) = \begin{cases} F(2 * X) \text{ DIV } 2 & \text{If } X < 30 \\ 3 * X & \text{If } X \geq 30 \end{cases}$$

- F is recursive, since it calls itself. For example:

$$\begin{aligned} F(50) &= F(50 \text{ DIV } 3) - 1 \\ &= F(16) - 1 \\ &= (G(32) + 4) - 1 = (96 + 4) - 1 = 99 \end{aligned}$$

- Although it appears that G is not recursive, it actually calls itself indirectly by invoking F, which then invokes back G. For example:

$$\begin{aligned} G(23) &= F(46) \text{ DIV } 2 \\ &= (F(15) - 1) \text{ DIV } 2 \\ &= ((G(30) + 4) - 1) \text{ DIV } 2 \\ &= ((90 + 4) - 1) \text{ DIV } 2 \\ &= 93 \text{ DIV } 2 = 46 \end{aligned}$$

- Such functions are indirectly recursive, or mutually recursive.

Aim: How can we write functions that are indirectly recursive?

Development:

1. A subprogram is indirectly recursive (mutually recursive) when it calls another subprogram that calls back the first subprogram.

2. Examine the following program, which evaluates F and G for an input value of X. Why wouldn't the program compile correctly? (Elicit responses.)

```
PROGRAM FandG (input, output);
VAR
  X : integer;

FUNCTION F (X : integer) : integer;
BEGIN
  write ('F');
  IF X<= 20 THEN
    F := G (2 * X) + 4
  ELSE
    F := F (X DIV 3) - 1
END; (* F *)

FUNCTION G (X : integer) : integer;
BEGIN
  write ('G');
  IF X< 30 THEN
    G := F (2 * X) DIV 2
  ELSE
    G := 3 * X
END; (* G *)

BEGIN (* FandG *)
  write ('Enter an X value: ');
  readln (X);
  writeln (F(X) : 8, ' = F(', X : 1,')');
  writeln (G(X) : 8, ' = G(', X : 1,')')
END.  (* FandG *)
```

This program would not compile correctly because function F calls function G before G has been defined.

3. How can this problem be corrected? A <u>forward reference</u> can be used. After the VAR statement in program FandG, add the statements:

```
FUNCTION G (X : integer) : integer; FORWARD;
```

and change the function heading for G to FUNCTION G. A typical run might be:

```
Enter an X value: 10
FGFFGFFG      31 = F(10)
GFG      62 = G(10)
```

5. Indirect recursion is often used in programs that calculate the value of arithmetic expressions, because of the recursive nature of mathematical definitions. Terms are defined using factors. Factors in turn are defined using expressions, and expressions are defined using terms.

6. Let's write a program that uses two indirectly recursive functions, Sum and Prod, to return the value of this expression for any positive integral value of N:

(... (((1 + 2) * 3 + 4) * 5 + 6) * 7 + ... + (N - 1) * N)

- How will the main program know whether to call Prod or Sum for a particular value of N? (Elicit responses.) Since we always multiply by an odd number and add an even number, the program can easily decide which operation is to be performed first.

<u>Possible solution:</u>

```
PROGRAM Calculate (input, output);
VAR
  N : integer;

FUNCTION Sum (N : integer) : integer; FORWARD;

FUNCTION Prod (N : integer) : integer;
BEGIN
  IF N = 1 THEN
    Prod := 1
  ELSE
    Prod := Sum (N - 1) * N
END; (* Prod *)

FUNCTION Sum;
BEGIN
  Sum := Prod (N - 1) + N
END; (* Sum *)

BEGIN ( * Calculate * )
  write ('Enter a positive integral value of N: ');
  readln (N);
  IF N < 0 THEN
    writeln ('Incorrect input')
  ELSE IF (N MOD 2 = 0) THEN
    writeln (Sum (N))
  ELSE
    writeln (Prod (N))
END.  ( * Calculate * )
```

7. After working through several examples of recursion by hand, one realizes the tremendous amount of bookkeeping performed by the computer. How does the computer do this? The computer system maintains a special set of contiguous memory locations called a <u>stack</u>. In general, each time a function is called recursively, a new set of local variables and intermediate values are pushed on the stack along with the memory address of the point of call. Any reference to a local variable is through the current top of the stack. When the function returns, the stack is popped, the allocated values at the top of the stack are lost,

and the previous set of values become the current top of the stack. If there are a large number of recursive calls, the stack overflows and an error results. (This mechanism will be examined more closely in later lessons.)

Summary:

1. What is an indirectly recursive function?

2. Why must forward references be used with indirect recursion?

3. How does the computer use stacks to keep track of recursive calls?

Exercises:

1. Show the output for program FandG when $X = 40$ and $X = 3$.

2. Modify program Calculate so that it calculates expressions of the form:

$$((\ \ldots\ (((((1 * 2) + 3) * 4) + 5) * 6) + 7\ \ldots\ * (N - 1) + N)$$

3. The binomial coefficients:

$$C(N, K) = \frac{N!}{K!\,(N - K)!}$$

can be defined recursively by the formula $C(N, K) = C(N - 1, K - 1) + C(N - 1, K)$. Prove this formula and write a recursive function that calculates these coefficients.

Topic 35

TOWERS OF HANOI PROBLEM

Performance Objectives: Students will be able to:

- state the Towers of Hanoi problem
- show how the problem leads to a recursive solution
- provide the recursive calls from any given state in a solution

Motivation:

- A Far-Eastern legend relates that there was a temple in Hanoi which contained three posts. On the first post there was a tower of 64 disks of graduated sizes. The priests at the temple were ordered to transfer all of the disks from the first to the third post according to the following two rules:

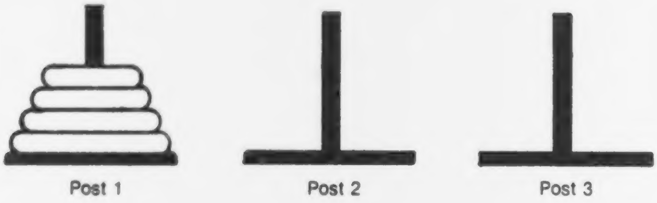
  - Only one disk at a time could be transferred from post to post.
  - A larger disk was never to be placed on top of a smaller disk.

  As an incentive for speedy execution, the priests were promised that they would obtain Nirvana at the conclusion of the task.

Aim: How can the computer be used to solve the Towers of Hanoi problem?

Development:

1. Have students demonstrate a solution to a smaller version of the towers problem using a wooden or plastic model that can be easily constructed or purchased in a toy store:



   The need for a well defined algorithm will evolve as students grope for a solution using only four or five disks.

2. When we start to transfer the smallest disk from the top of one post, the question arises, "to where?" Once students realize that transferring a tower of 64 disks depends on the ability to transfer a tower of 63 disks, a recursive solution seems natural.

3. Suppose that the first post from which disks are to be removed is denoted by startpost, the post to which the disks are to be moved is denoted by endpost, and the third post is denoted as freepost. The recursive algorithm for moving N disks from the startpost to the endpost can be stated in the following three steps:

   - Move the top (N - 1) disks from the startpost to the freepost.
   - Move the largest disk from the startpost to the endpost.
   - Move the (N - 1) disks from the freepost to the endpost.

4.

   let Movetower (N, A, B)

   represents the task of moving a tower of N disks from post A to post B.

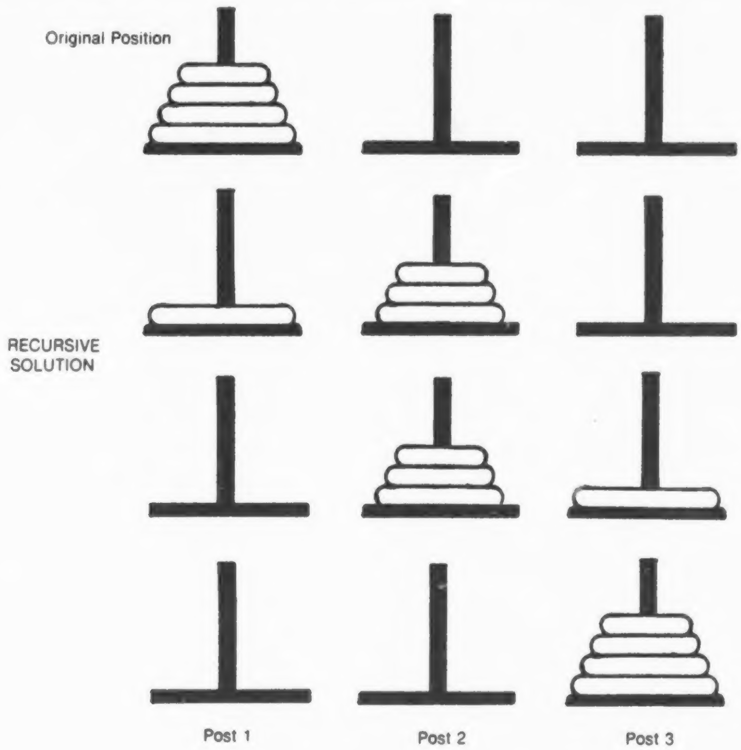
   let Movedisk (A, B)

   represents the task of moving a disk from post A to post B. (Note that A and B can only be 1, 2, or 3.)

5. The algorithm for moving N disks from a startpost to an endpost can now be stated more succinctly:

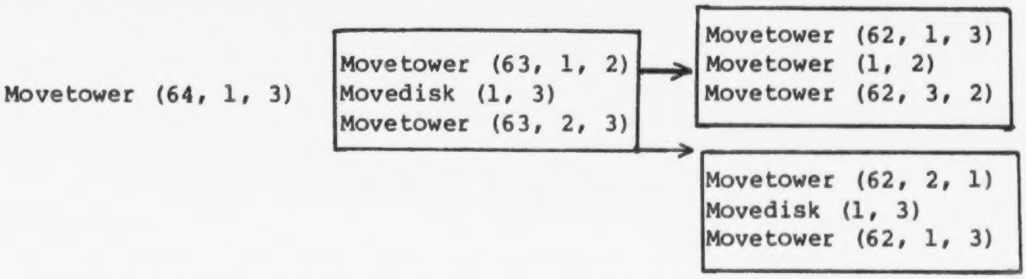
```
Movetower (N, startpost, endpost)

- Movetower (N - 1, startpost, freepost)
- Movedisk (startpost, endpost)
- Movetower (N - 1, freepost, endpost)
```

6. The diagram below illustrates a recursive solution for four disks:



```
Movetower (4, 1, 3) = { Movetower (3, 1, 2)
                        Movedisk (1, 3)
                        Movetower (3, 2, 3)
```

7. The Tower of Hanoi problem can now be stated as Movetower (64, 1, 3). Several recursive calls are illustrated below:


Movetower (64, 1, 3)
Movetower (63, 1, 2)
Movedisk (1, 3)
Movetower (63, 2, 3)
Movetower (62, 1, 3)
Movetower (1, 2)
Movetower (62, 3, 2)
Movetower (62, 2, 1)
Movedisk (1, 3)
Movetower (62, 1, 3)

8. What is the simple case that ends the recursive calls? (Elicit responses.) When we move a tower of one. For example:

    Movetower (1, 1, 3)

    is equivalent to:

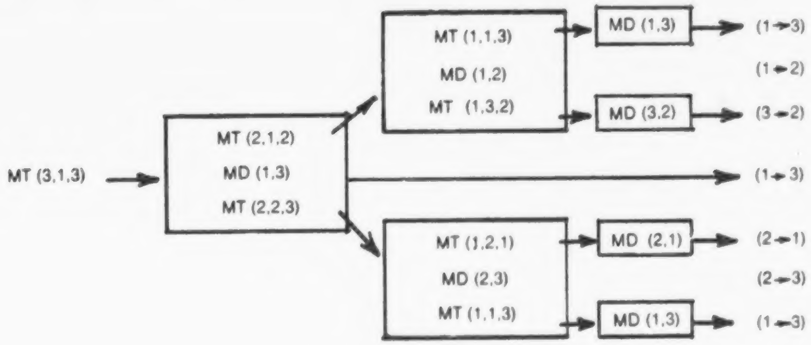
    Movedisk (1, 3)

9. A complete program for solving the problem can now be written. (Note: Since the sum of the posts is six, the freepost can always be determined by subtracting the startpost and endpost from six. For example, if startpost = 1 and endpost = 3, then freepost = 6 - 1 - 3 = 2.)

```
PROGRAM Hanoi (input, output);
VAR
  numdisks : integer;

PROCEDURE Movetower (numdisks, startpost, endpost : integer);
VAR
  freepost : integer;

PROCEDURE Movedisk (takeoff, puton : integer);
BEGIN
  writeln (takeoff, '->', puton)
END; (* Movedisk *)
```

```
BEGIN (* Movetower *)
  freepost := 6 - startpost - endpost;
  If numdisks = 1 THEN
    Movedisk (startpost, endpost)
  ELSE
    BEGIN
      Movetower (numdisks - 1, startpost, freepost);
      Movedisk (startpost, endpost);
      Movetower (numdisks - 1, freepost, endpost)
    END (* ELSE *)
END; (* Movetower *)

BEGIN (* Hanoi *)
  read (numdisks);
  writeln ('enter number of disks to be moved');
  Movetower (numdisks, 1, 3)
END.  ( * Hanoi * )
```

10. This is the diagram for a complete recursive solution when numdisks = 3. (MT = Movetower, MD = Movedisk.)



11. Since every recursive call breaks down into two recursive calls until the simple case is reached, the number of transfers involved in moving N disks is $2^N - 1$. Thus $(2^3 - 1)$, or seven, moves are needed to move three disks. Even if a computer can do one billion calculations a second, it would still take several billion years to move 64 disks.

Summary:

1. What are the two rules in the Tower of Hanoi problem?

2. What would be the next two recursive calls to Movetower (48,3,2)?

Exercises:

1. Show a complete solution to the Towers of Hanoi problem for four disks. How many moves would be needed?

2. Write the next two levels of recursion for Movetower (24, 3, 1).

Topic 36

ITERATION VS. RECURSION

Performance Objectives: Students will be able to:

- define several classes of algorithms
- explain the difference between iterative algorithms and recursive algorithms
- write recursive functions iteratively and iterative functions recursively

Vocabulary: iterative algorithm, recursive algorithm, sequential algorithm

Motivation:

- Using the definition of N!, write a nonrecursive function to determine factorials. (Elicit responses.)

- This is one solution to the problem:

```
FUNCTION Fact (N : integer) : integer;
VAR
  I, factorial : integer;
BEGIN
  factorial := 1;
  FOR I := 2 TO N DO
    factorial := factorial * I;
  Fact := factorial
END; (* Fact *)
```

- If we compare this iterative algorithm with the recursive algorithm we wrote previously, which is better? Why?

Aim: What is the difference between iteration and recursion?

Development:

1. Throughout the term we have emphasized the need for well defined algorithms before any code is written. We have already encountered several types of algorithms:

   - sequential algorithms, which proceed toward a solution one step at a time (and are used most frequently)
   - iterative algorithms, which repeat a series of steps that converge to a solution of the problem (e.g., the program segment above for function Fact)
   - recursive algorithms, which break a problem into smaller instances of the same problem until simple case(s) can be solved and reassembled to produce a solution to the original problem (e.g., the Fibonacci sequence in Topic 33)

2. The type of algorithm we use is often determined by the nature of the problem. Though it can be shown that all recursive algorithms can be written iteratively, recursion is often the more inherently plausible solution, such as in the Towers of Hanoi problem. In most practical problems we will encounter, space and time considerations are important. Recursion uses a great deal of memory to store copies of local variables, and it takes an excessive amount of time to allocate and deallocate storage. It is not used frequently when an iterative solution can be easily formulated.

3. The euclidean algorithm will help us illustrate the differences between programs written iteratively and recursively. The euclidean algorithm is a technique for finding the greatest common divisor (gcd) of two positive integers. (Note: See any college algebra textbook for more information.) It uses the theorem that the gcd of A and B is also the gcd of B and the remainder of A/B. (Recall that if B @= A then A = B x Q + R, where Q and R are the quotient and remainder, respectively, of A/B.)

   . To find the gcd of 68 and 12:

dividend (A)	=	quotient	x	divisor (B)	+	remainder (R)
68	=	5	x	12	+	8
12	=	1	x	8	+	4
8	=	2	x	4	+	0
				↑ gcd		

   Since the remainder is 0, the gcd is 4.

   . How would we write iterative and recursive functions that use the euclidian algorithm to find the greatest common divisor of two positive integers? (Elicit responses.)

   Possible iterative solution:

```
FUNCTION Gcd (A, B : integer) : integer; (* iteratively *)
  VAR
    rem : integer;
  BEGIN
    WHILE B<>0 DO
      BEGIN
        rem := A MOD B;
        A := B;
        B := rem
      END; (* WHILE *)
    Gcd := A
  END;
```

Possible recursive solution:

```
FUNCTION Gcd (A, B : integer) : integer; (* recursively *)
  BEGIN
    IF B = O THEN
      Gcd := A
    ELSE
      Gcd := Gcd (B, A MOD B)
  END; (* Gcd *)
```

Note: The greatest common divisor is the same as the highest common factor.

Applications:

1. Write an iterative function that finds the Nth term of the Fibonacci sequence.

Possible solution:

```
FUNCTION Fibonacci (N : integer) : integer;
VAR
  I, old, new, next : integer;
BEGIN (* Fibonacci *)
  IF (N = 2) OR (N = 1) THEN
    Fibonacci := 1
  ELSE
    BEGIN
      old := 1;
      new := 1;
      FOR I := 3 TO N DO
        BEGIN
          next := old + new;
          old := new;
          new := next
        END;
      Fibonacci := next
    END (* ELSE *)
END; (* Fibonacci *)
```

2. Write iterative and recursive functions that find values of hermite polynomials, which are defined as follows:

$$H_N(X) = \begin{cases} 1 & \text{IF } N = 0 \\ 2X & \text{IF } N = 1 \\ 2XH_{N-1}(X) - 2(N-1)H_{N-2}(X) & \text{IF } N > 1 \end{cases}$$

Possible recursive solution:

```
FUNCTION H (N : integer; X : real) : real; (* recursively *)
BEGIN
  IF N = 0 THEN
    H := 1
  ELSE IF N = 1 THEN
    H := 2 * X
  ELSE
    H := 2 * X * H (N - 1, X) - 2 * (N - 1) * H(N - 2, X)
END; (* H *)
```

Possible iterative solution:

```
FUNCTION H (N : integer; X : real) : real; (* iteratively *)
VAR
  old, new, next : real;
  I : integer;
BEGIN
  IF N = 0 THEN
    H := 1
  ELSE IF N = 1 THEN
    H := 2 * X
  ELSE
    BEGIN
      old := 1;
      new := 2 * X;
      FOR I := 2 TO N DO
        BEGIN
          next := 2 * X * new - 2 * (I - 1) * old;
          old := new;
          new := next
        END; (* FOR *)
      H := next
    END (* ELSE *)
END; (* H *)
```

Summary:

1. What is the difference between recursive and iterative algorithms?

2. When would a recursive algorithm be preferable? When would an iterative algorithm be preferable?

Exercises:

1. Write an iterative function to find a binomial coefficient.

Hint: $C(N, K) = \dfrac{N \times (N-1) \times (N-2) \times \ldots (N-(K-1))}{1 \times 2 \times 3 \times \ldots K}$

2. Write a program that uses the function in Exercise 1 to draw a Pascal triangle.

3. Write iterative and recursive functions that find values of the Legendre polynomials $L_N(X)$:

$$L_N(X) = \begin{cases} 1 & \text{IF } N = 0 \\ X & \text{IF } N = 1 \\ ((2N - 1) \times L_{N-1}(X) - (N-1) L_{N-2}(X))/N & \text{IF } N > 1 \end{cases}$$

4. Write a recursive function that calculates values of Ackermann's function ACK(M, N) where $M >= 0$, $N >= 0$:

$$ACK(M, N) = \begin{cases} N + 1 & \text{IF } M = 0 \\ ACK(M - 1, 1) & \text{IF } N = 0 \\ ACK(M - 1, ACK(M, N-1)) & \text{IF } M > 0 \text{ and } N > 0 \end{cases}$$

Topic 37
USER-DEFINED SCALAR TYPES

Performance Objectives: Students will be able to:

- create user-defined scalar types
- explain how user-defined scalar types are used in control and assignment statements
- apply certain functions to user-defined scalars
- list major limitations of user-defined scalars

Vocabulary: scalar, user-defined scalar type, enumeration types, type definition, ordinal, type, pred, ord, succ

Motivation:

- What will be the output of this program? (Elicit responses.)

```
PROGRAM Scalars (output);
TYPE grade = (freshman, sophomore, junior, senior);
VAR
  student : grade;
BEGIN
  student := junior;
  IF student > junior THEN
    writeln ('This student should graduate this year.')
END.  (* Scalars *)
```

  Since the student is only a junior, there will be no output from program Scalars.

- What are the four types of values a Pascal variable may hold?

  - Integer, real, character, and boolean. These are referred to as the standard types, or scalar types, and have the following properties:

    (1) The values are ordered:

    (a) 6 > 4
    (b) 6.1 < 7.0
    (c) 'B' > 'A'
    (d) TRUE > FALSE

    (2) The values are also distinct from other values of the same or different types.

- What variable is used in program Scalars?

  - student

- . What type is the variable?

  - grade

- . Is this one of Pascal's standard types?

  - no

- . In the program Scalars, what are two values for student that are greater than sophomore?

  - junior and senior

Aim: How do we write a program with user-defined scalar types?

Development:

1. The values for type grade are freshman, sophomore, junior, and senior. User-defined types such as grade are also referred to as enumeration types, because their values are listed and ordered. The ordered value of freshman is 0, with a value of 3 for senior.

2. How does one create a new scalar type?

   - In the declaration part of the program between CONST and VAR, write the word TYPE.
   - Create an identifier for the new type.
   - Write an equal sign.
   - List the values, in parentheses, separated by commas.

3. This is the syntax for a user-defined scalar type:

```
TYPE
  identifier = (valuel, value2, ...);
```

   . Write a type declaration for day to include the seven days of the week.

```
- TYPE
    day = (Mon, Tues, Wed, Th, Fri, Sat, Sun);
```

   . Is Sat > Fri?

   - Yes, the relational operators are dependent on the order of the enumerated values.

   . Write the variable declaration for workday to be of type day.

```
- VAR
    workday : day;
```

   . Write a statement that assigns Monday to workday.

```
- workday := Mon;
```

4. The only way a user-defined scalar type can be assigned a value is from an assignment statement within the program. Unfortunately, Pascal does not permit assignment from read statements or direct output from write statements. Therefore these statements would be illegal:

```
(a) read (workday)
(b) write (workday)
```

However, user-defined types can be used in control statements. The following statements are valid:

```
(a) IF workday = Sun THEN
      write ('I hope she at least gets Saturday off.');

(b) CASE workday OF
      Mon, Tues, Wed, Th, Fri : writeln ('That is normal.');
      Sat, Sun : writeln ('Overtime is in order.')
    END;

(c) FOR workday := Mon TO Fri DO
      BEGIN
        read (wages);
        salary := salary + wages
      END;
```

Enumeration types may also be used to test conditions in WHILE and REPEAT statements.

5. The CASE structure offers a solution to the illegal use of write (workday):

```
CASE workday OF
  Mon : write ('Monday');
  .
  .
  .
  Fri : write ('Friday');
END;
```

6. Pascal's built-in functions--ord, pred, and succ--are valid with user-defined scalar types:

```
(a) ord (Mon) = 0;
(b) succ (Th) = Fri;
(c) pred (Sun) = Sat;
```

Attempting to evaluate the succ of the last item or the pred of the first item would result in an error, since these values are undefined.

Summary:

1. What are the four standard data types?

2. How can we create other data types besides the four standard types?

3. Where are user-defined types declared in a program?

4. What operators and functions can be used with user-defined types?

Exercises:

1. Assume Pascal does not define the type boolean. Write a user-defined type that would define boolean to be the set of values true and false.

   Answer: TYPE boolean = (false, true);

2. Given the following user-defined type, evaluate the functions that follow:

   ```
   TYPE rank = (captain, major, colonel, general);
   ```

   (a) ord (captain)
   (b) succ (colonel)
   (c) pred (major)
   (d) succ (general)

   Answers: (a) 0, (b) general, (c) captain, (d) undefined

3. Write a program with the following choices:

   0. red
   1. orange
   2. yellow

   First ask the operator to select a color by number, and then print the color corresponding to the number selected. Create a function Hue that incorporates a CASE statement to assign the appropriate color to Hue.

4. Write a program to read in the inches of rainfall for each of the 12 months. Declare type month. Output the total annual rainfall.

Topic 38
SUBRANGE TYPES

Performance Objectives: Students will be able to:

- write subrange type declarations
- state the reasons for using subranges

Vocabulary: subrange type, type compatibility, assignment compatibility

Motivation:

- What is the purpose of the type declaration below?

```
PROGRAM Subrange (input, output);
TYPE
  digit = 0..9;
VAR
  X : digit;
BEGIN
  writeln ('Enter a nonnegative integer');
  write ('less than 10 ');
  readln (X);
  writeln ('Yes ', X, ' is less than 10')
END. (* Subrange *)
```

- The type declaration insures that any variable of type digit will be an integer from 0 to 9. If the variable (X in this case) is not in the subrange, an error message will appear.

Aim: How do we use subranges to restrict the range of variables in programs?

Development:

1. Since we know that the variable X in program Subrange must be an integer from 0 to 9, we can restrict X to an integer in this range by:

   - declaring an identifier in the TYPE declaration to be in the desired range (as done above)
   - declaring a variable to be of this TYPE in the variable declaration block (see step 6)

2. Write the declarations necessary to restrict ch to the letters in the first half of the alphabet. (Elicit responses.)

Answer:

```
TYPE
  alphahalf = 'A'..'M';
VAR
  ch : alphahalf;
```

Note that in the type declaration, the lower bound 'A' is separated by two dots from 'M', the upper bound of the range. This declares alphahalf to include letters from 'A' to 'M,' inclusive.

3. The subrange feature is used so that the computer will inform the user, through an error message, when the integrity of the data has become suspect. It also allows others working on a program to get an idea of appropriate values for the program.

4. We have seen how subranges are used with characters and integers. They may also be used with user-defined scalar types. For example, write a TYPE declaration block to:

   - create day as a user-defined scalar for all seven days of the week
   - declare weekday to be a subrange consisting of the five work days Monday through Friday
   - declare weekend a subrange consisting of Saturday and Sunday

   Answer:

```
TYPE
  day = (Mon, Tues, Wed, Th, Fri, Sat, Sun);
  weekday = Mon..Fri;
  weekend = Sat..Sun;
```

   Note that only the scalars for the user-defined type day are enclosed in parentheses. The upper and lower bounds of the subrange types are separated by two dots.

5. We can now declare variables to be of these types:

```
VAR
  workingday : weekday;
  dayoff : weekend;
```

   Although a scalar like Fri may appear in only one user-defined type, it may appear in more than one subrange type. That is, we can add to the following types above:

```
longweekend = Fri..Sun;
```

6. You may declare variables to be of a subrange type directly without a type declaration. Although this is legal, it is not recommended. The following is such a declaration:

```
VAR
  dayoff : Sat..Sun;
  X : 0..9;
```

7. Procedure parameters and functions must be declared to be of a certain type, and the type must be declared the long way. The shorthand designation for parameters and functions of the subrange type cannot be used, as they must be declared to be of a type name. For example:

```
FUNCTION sum (A, B : integer) : 1..1000;
```

is illegal.

```
FUNCTION sum (A, B: integer) : uptothousand;
```

is legal if we had this type declaration:

```
TYPE
  uptothousand = 1..1000;
```

8. The assignment

```
A := C
```

is valid if A and C are assignment compatible. That is, they must either be type compatible, or A can be type real when C is type integer. To be type compatible, A and C must be the same type, or expression C must fall in the subrange of A. For example, if we use the following variable declaration block:

```
VAR
    loval : 1..64;
    hival : 1..100;
```

it would always be legal to let

```
hival := loval
```

The assignment

```
loval := hival,
```

however, would not be compatible because values for hival fall outside the range of loval. Even though these 2 are not assignment compatible, no error will result as long as hival has a value between 1 and 64.

Summary:

1. What are some advantages of subrange types?

2. What special symbol is used to specify the range of a subrange?

Exercises:

1. One way to declare satscore would be:

```
VAR
  satscore : integer;
```

   Since we know in what range satscore must lie (200-800), write a more appropriate subrange declaration for satscore.

2. The procedure Range has a value parameter, X, that is integral and lies in the range 50 to 100. Write the definitions and declarations necessary to restrict X to the appropriate subrange.

3. Create a user-defined scalar, cardspot, that is thirteen cards of any suit. Write a type definition subrange for facecard.

Appendix A
RESERVED WORDS

The following words are reserved in standard Pascal; that is, they may not be used as variable names in a Pascal program. In this publication, all reserved words have been printed in capital letters.

AND
ARRAY
BEGIN
CASE
CONST
DIV
DO
DOWNTO
ELSE
END
FILE
FOR
FUNCTION
GOTO
IF
IN
LABEL
MOD
NIL
NOT
OF
OR
PACKED
PROCEDURE
PROGRAM
RECORD
REPEAT
SET
THEN
TO
TYPE
UNTIL
VAR
WHILE
WITH

Appendix B
STANDARD CONSTANTS, FILES, AND TYPES

CONSTANTS

true

false

maxint

FILES

input

output

TYPES

integer

real

char

boolean

text

Appendix C
STANDARD FUNCTIONS

NAME	FUNCTION	TYPE OF ARGUMENT	TYPE RETURNED
abs(X)	absolute value	real or integer	same as argument
arctan(X)	angle whose tangent is X	real or integer	real (in radians)
chr(X)	character whose ordinal number is X	integer	character
cos(X)	cosine	real or integer (X is in radians)	real
eof(F)	end of file test of F	file	boolean
eoln(F)	end of line test of F	file	boolean
exp(X)	e to the power x	real or integer	real
ln(X)	natural log of X	real or integer	real
odd(X)	odd test of X	integer	boolean
ord(X)	ordinal number of X	any ordinal type	integer
pred(X)	the predecessor of X	any ordinal type	same as argument
round(X)	rounding off	real	integer
sin(X)	sine	real or integer (X is in radians)	real
sqr(X)	square of X	real or integer	same as argument
sqrt(X)	square root of X	real or integer	real
succ(X)	successor of X	any ordinal type	same as argument
trunc(X)	truncation	real	integer

Appendix D

DIFFERENCES BETWEEN STANDARD PASCAL AND UCSD PASCAL

Differences between standard Pascal and UCSD Pascal

UCSD Pascal	Standard Pascal
Program Heading	
No file parameters need be listed in the program heading.	All input/output external files must be listed as parameters in the program heading.
Files	
A special interactive file called KEYBOARD is provided to prevent the writing on the screen of data entered at the keyboard. A procedure called SEEK permits the program to randomly access file components.	Only sequential access files are allowed.
Case Statement	
Execution continues when the CASE variable does not match any of the labels.	An error results when the CASE variable does not match any of the labels.
Strings	
A special string data type is provided along with string functions and procedures (e.g., length, pos, insert).	String processing is usually accomplished using packed arrays of character.
Subprogram parameters	
Procedures and functions cannot be used as parameters.	Procedures and functions can be used as parameters in other subprograms.
Pack and Unpack Procedures	
Though PACK and UNPACK are not implemented, these operations are automatically performed by the operating system.	PACK and UNPACK procedures are provided to transfer data between packed and unpacked arrays.

UCSD Pascal	Standard Pascal
Exit Procedure	
An EXIT procedure is used for transferring out of a block.	A GOTO is permitted to a label outside the block in which the GOTO is written.
Dispose Procedure	
The DISPOSE procedure is not provided.	DISPOSE is used to return dynamically allocated memory for future reallocation.

Appendix E
OPERATOR PRECEDENCE

1. Parentheses are executed first.

2. If operators or equal precedence are encountered, they are executed in order from left to right.

3. In order, from highest precedence to lowest, the operators are:

   (a) NOT

   (b) * / DIV MOD AND

   (c) + - OR

   (d) = <= >= <> > < IN

Appendix F
OPERATING SYSTEMS

Part I--TRS-80 Pascal

TRS-80 Pascal can be used with the following TRS-80 computers:

- Model I--48K, two disk drives
- Model III--48K, one disk drive (for TRS DOS 1.3 users); two disk drives (for LDOS operating system users, catalog #26-2214)
- Model IV--two disk drives

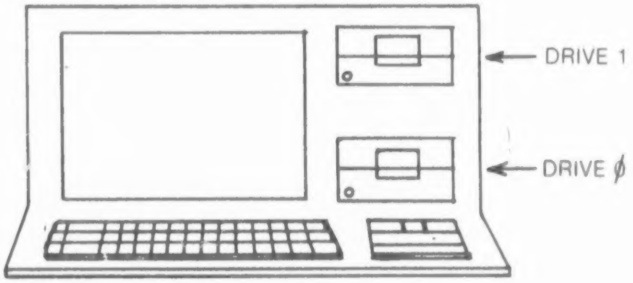
Turning on the Computer and Booting Up

- On the side of the computer, under the keyboard there is a power switch. Before turning the power on, make sure the doors of the drive(s) are shut. Then turn the power on. You will see the red light of the disk drive turn on and then off.

- When the light turns off, open the drive door, and gently push the disk into the drive. Shut the door, and then press the orange <reset>key. The red light will turn on and then off. When the light is on, the machine is being booted up and loading into memory.

Making a Backup Copy of TRS80-Pascal

A. Making backup copies on Model III and Model IV two-drive systems



TRS80 Pascal Model III version contains two diskettes, labeled 1 of 2 and 2 of 2. These diskettes can be used with TRS 80 Model III and Model IV computers. It is important to make backup copies of these master diskettes:

1. Turn the computer on. You will see the light on drive 0 go on and then off.

2. Insert the master disk labeled 1 of 2 in drive 0 and press the orange <reset>button. The light on drive 0 will go on and then off.

3. A prompt will appear on the screen with this message:

    Enter Date (MM/DD/YY)?
    - Type: the date, remembering to use two digits for MM and DD. For example, January would be 01 not 1.
    - Press: <enter>

    Enter Time (HH:MM:SS)?
    - Press: <enter>.

    You will then see the message TRSDOS READY.

4. Place a new blank disk in drive 1 and type: Backup<enter>.

5. The following prompts will appear:

    Source Drive Number?
    - Type: 0<enter>

    Destination Drive Number?
    - Type: 1<enter>

    Source Disk Master Password?
    - Type: PASSWORD<enter>

    The backup process is now in session. The lights will be alternating on and off on drives 0 and 1. The computer is reading, verifying, and writing all the files from the master disk in drive 0 to the blank disk in drive 1. When the backup process is complete, the following message will be displayed on the screen:

    ** BACKUP COMPLETE **

    This will be followed by the message TRSDOS READY.

6. Take both diskettes out of the drives, and label the backup copy TRS80 Pascal 1 of 2.

7. Insert the master diskette labeled TRS80 Pascal 2 of 2 in drive 0, and a new blank diskette in drive 1. Repeat steps 4 and 5. When the backup is complete, follow step 6, but label the backup copy TRS80 Pascal 2 of 2.

8. To be absolutely sure that the backup copies work, boot up the machine using these copies.

B. Making backup copies on a Model III one-drive system

1. Turn on the computer. The light on the drive will go on and then off.

2. Place the master disk labeled 1 of 2 in the drive and press the <reset>key.

3. Follow step 3 in A.

4. Follow step 4 in A.

5. The following prompts will appear:

   Source Drive Number?
   - Type: 0<enter>
   Destination Drive Number?
   - Type: 0<enter>
   Source Disk Master Password?
   - Type: PASSWORD<enter>

6. A prompt will appear telling you to insert the destination diskette. (The destination diskette is the new blank diskette that you want to copy onto.) Take out the master and insert the blank disk into the drive. Press<enter>. The new disk will now be formatted. A prompt will appear to insert the source diskette (master diskette). Take out the blank disk, insert the master disk, and then press<enter>.

7. The backup process will now begin. Prompts will appear alternately to insert the source diskette (master) and the destination diskette (blank diskette). After each insertion, press <enter>. This will be done several times until the following message appears:

   ** BACKUP COMPLETE **

   The will be followed by the message TRSDOS READY.

8. Repeat steps 4 through 7 using the master disk labeled 2 of 2.

C. Making backup copies on a Model I two-drive system

. Model I does not look like Model IV. TRS80 Pascal Model I Version contains 3 diskettes labeled 1 of 3, 2 of 3, and 3 of 3.

1. Turn on the computer. You will see the light on drive 1 go on and then turn off.

2. Place the disk labeled 1 of 3 in drive 1.

3. Place the TRSDOS 2.3 operating system diskette in drive 0 and press the <reset> button.

4. DOS READY will appear on the screen. Then type: BACKUP<enter>

5. Remove the TRSDOS 2.3 disk in drive 0 and replace it with a new blank diskette.

The following prompts will appear:

Source Drive Number?
- Type: 1<enter>

Destination Drive Number?
- Type: 0<enter>

Backup Date (MM/DD/YY)?
- Enter: the month, date, and year, and then press<enter>. Remember that you must press two digits for each. January would be 01 not 1.

6. When the backup is complete the following message will appear on the screen:

   ** BACKUP COMPLETE **

   Then press<enter>to continue.

7. Repeat steps 3 through 6 for disks 2 of 3 and 3 of 3.

Entering, Compiling, and Executing a TRS80 Pascal Program

. Boot up the computer with the disk that contains the file ED/CMD. (In the Model III version it is disk 1 of 2.)

. The editor can be used to create a new program or to edit an existing program.

A. Creating a new program

1. Type: ED<enter> (This loads the editing program into the computer's memory.)

2. When the editor is loaded, the symbols *EOB will appear in the upper left-hand corner of the screen. (EOB stands for "end of buffer.")

   When creating a new file, the buffer is empty. Before the new file (program) can be entered into the buffer, you must insert blank lines into the buffer, either a few at a time or one at a time. Blank lines are inserted by pressing the<shift>and<@>keys simultaneously. After a blank line is inserted, a program can be entered.

   Press<shift-@>four times and type in the following program using the <→> and <←> keys (left arrow and right arrow keys) for indentation.

```
PROGRAM FIRST;<enter>
BEGIN<enter>
  WRITELN ('MY FIRST PROGRAM')<enter>
END. <enter>
*EOB
```

*EOB remains on the screen but now follows the program you have entered.

Notice that every time<enter>is pressed, the cursor appears at the beginning of the next line.

Once the text is entered, press<clear>and then the letter C to enter the command mode. The following prompt will appear at the bottom of the screen: < >.

3. To save the text in the editor, type: EXIT <enter>. The following prompt will appear: <EXIT>FILE:. Type in a valid file name followed by the extension/PCL including a drive specification, if desired. For example, if you wanted to name the file First, at the command: <EXIT> FILE:, you would type: FIRST/PCL <enter>. If you wanted to save it in drive 1, you would type: FIRST/PCL: 1 <enter>.

   If a drive is not specified, the system will default to the lowest numbered drive.

   Note that executing the exit command will save the edited text to a file. After the file is saved the following will appear on the screen:

```
RENAME FILE
STACK USED = 3708 OF 4032     HEAP USED = 1228 of 15924
TRSDOS Ready
```

   At this point you can compile the program. (See below.)

4. To terminate the editing session without saving the text, type: QUIT when the following prompt appears: < > QUIT <enter>.

5. Now that the program has been saved it can be compiled. Enter:

```
PASCAL FIRST <enter>
         ↑
   name of file
```

   This command will load and execute the Pascal compiler, open the file First/PCL, and read the file one line at a time. The compiler will write the object code to First/OBJ and send a listing to the screen with error messages. The following listing would be sent to the screen for the program First:

```
TRS80 PASCAL VER:  02.00.00 13000000 00:06:43 05/18/83 PA

    1   PROGRAM FIRST;
    2   BEGIN
    3     WRITELN ('MY FIRST PROGRAM')
    4   END.
FIRST
  NO ERRORS DETECTED
NO ERRORS DETECTED
STACK USED = 1715 of 4096     HEAP USED = 2828 of 6232
TRSDOS Ready
```

Now that the program has been compiled with no errors, it can be executed. When you see the TRSDOS Ready prompt, type: RUN FIRST<enter>.

You will now see two prompts:

```
Input = <enter>
Output =<enter>
```

By pressing <enter> at both prompt messages, input will be received by the keyboard and output will be displayed on the CRT. (See the manual for any other input/output devices.)

The writeln causes MY FIRST PROGRAM to appear on the screen with the amount of stack and heap used. (See the system implementation manual for more information on stacks and heaps.)

6. To edit an existing program, load the program First/PCL into the editor so that changes can be made. Type: ED FIRST/PCL<enter>. You will see the program listing on the screen with the cursor over the <u>P</u> in PROGRAM.

   Try these cursor movements:

   - <↑> moves cursor up a line
   - <↓> moves cursor down a line
   - <→> moves cursor to right
   - <←> moves cursor to left
   - <enter> moves cursor to beginning of next line
   - <clear> <←> moves cursor to beginning of line
   - <clear> <→> moves cursor to end of line
   - <clear> <↑> scrolls the display one page towards the beginning of the file
   - <clear> <↓> scrolls the display one page towards the end of the file
   - <clear><H> positions cursor home (top left of screen)

These keys allow you to insert and delete text:

- <shift><→> deletes character under the cursor
- <shift><←> deletes the entire line under cursor
- <clear><K> deletes text from cursor to the end of line
- <clear><I> inserts text. Place the cursor at the desired location, press <clear><I>, and type in text.

  <shift><@> inserts a blank line into text

Try using text deletion and insertion to change the program to:

```
WRITELN ('MY PROGRAM')
```

Right now, the cursor is positioned over the P in PROGRAM.

- Press <enter> or <↓> two times so that the cursor is positioned at the beginning of the writeln line.
- Using the <→>, move the cursor over the F in FIRST.
- Delete FIRST by pressing <shift><→> five times to delete all the letters in FIRST and one more time to delete the extra space.
- After this is done, press <enter> and the cursor will be on the next line.

Now insert the word WORKS into the program:

```
WRITELN ('MY PROGRAM WORKS')
```

- Move the cursor over the second quote in the writeln statement.
- Press <clear><I> once. (Nothing will appear to happen.)
- Type in @WORKS. (You will notice that the text is inserted.)
- Press <enter>. The cursor is now on the next line and the quote is still there.

To insert the following line of text after BEGIN:

```
WRITE ('I AM PROUD');
```

- Move the cursor to the line where the text will be inserted. (Move it to the writeln.)
- Press <shift><@> to insert a blank line into the text buffer.
- Press <clear><I> to insert the write statement.
- Press <enter> after the entire insertion is made.

Now delete the entire writeln statement:

```
WRITELN ('MY PROGRAM WORKS')
```

- Move the cursor to the beginning of the writeln statement.
- Press <shift><←> and the entire writeln statement will disappear.
- Delete the ; in the write statement to make it a syntactically correct.

Since there is no need to save this program, practice the Quit command to leave the editor without saving the program.

- Press: <clear><C>. You will see the prompt <>.
- Enter: QUIT<enter>. You will see the prompt: Quit Really?
- Type: Y<enter>.

The editor has now been cleared.

Handling a program compiled with errors

1. Enter the following sample program* (By entering ED<enter> again):

```
PROGRAM TEST;
BEGIN
  WRITELN ('THIS IS MY FIRST PROGRAM.)
END.
*EOB
```

2. Save the program as TEST/PCL and then compile it. When the program is compiled, the following will appear on the screen:

```
TRS80 PASCAL VER:02.00.00 13000000 00:00:21 03/12/83

      1   PROGRAM TEST;
      2   BEGIN
      3     WRITELN ('THIS IS MY FIRST PROGRAM. )

                                                 202
*****
      4   END.

*****        20,   4,   13

      4 ERRORS DETECTED
      4  ')' EXPECTED
     13 'END' EXPECTED
     20 ',' EXPECTED
    202 STRING CONSTANT CANNOT SPAN LINES

STACK USED = 1715 OF 4096     HEAP USED = 3070 OF 6232

TRS DOS Ready
```

The compiler writes an error message on the line following the detected error. The error message line begins with 5 astericks. It also contains a pointer (∧) identifying the part of the line where the error was detected. Following the pointer (∧) there is an error code which replaces the type of error detected. After the listing, all generated errors are listed with their explanations.

When errors are detected in a program, the compiler creates a file called Pascal/ERR. This file contains the lines containing errors, along with the error messages.

*Program from Radio Shack Manual.

3. At this point, you can now edit the program using the editing commands, and then exit, compile, and run the program.

Using Other Special Characters

The following special characteristics may also be useful.

Sequence	Character
<clear><1>	[
<clear><2>	]
<clear><3>	∧
<clear><4>	{
<clear><5>	}
<clear><6>	¦
<clear><7>	—

Getting a Directory Listing

To get a directory listing, enter: DIR<enter>. If your system has one drive, DIR will give a directory listing of the disk in that drive. If your system has two drives, you can specify which drive you want listed by typing in the number of the drive:

```
DIR : 0<enter>    or    DIR : 1<enter>
```

Note: There must be a space before the colon in these two commands.

Not specifying a drive in a two-drive system causes the computer to default to the boot drive.

Deleting a File You No Longer Want

To delete a file you no longer want, enter: KILL "file name"<enter>. For example, to KILL TEST/PCL, you would enter: KILL TEST/PCL<enter>.

These are some of the more useful editing features. To learn how to use all the editing features, and to learn more about TRS80 Pascal, refer to the TRS80 Pascal manual that is supplied with the diskettes.

PART II--APPLE II and IBM

The following procedures are designed for a two-drive Apple II+ computer with standard diskettes (i.e., APPLE0:, APPLE1:, APPLE2:, and APPLE3:. Having nonstandard diskettes may require the use of a different diskette in a drive for certain procedures. The IBM personal computer has an almost identical Pascal system and any differences will be pointed out.

The name of a diskette in Apple Pascal and UCSD (University of California-San Diego) Pascal is always followed by a colon. For the two-drive Apple system we will need diskettes APPLE1:, APPLE2:, and APPLE3:.

The disk drives in Pascal are called #4: (for drive 1) and #5: (for drive 2). We may refer to a diskette by its name or the drive it is in. If APPLE1: is placed in drive 1 we may refer to this diskette as either APPLE1: or #4:.

In the material that follows we will use certain notation. The cursor on the screen will be denoted by an underscore. The word enter followed by a colon means that you are to type exactly what follows the colon on the computer. Brackets (i.e., [and]) will be used to enclose explanations of certain steps in the procedures. Whenever you are to perform specific actions on the computer or press special keys we will write these within the symbols <and>. For example, <cr> means for you to press the return (i.e., carriage return) key; <space bar> means press the space bar; <CTL-C> means hold down the control key while you press the letter C; and <turn off computer> means for you to turn off the computer.

Booting and Command Level

A. The process of getting the Apple II+ loaded with the Apple Pascal operating system is called booting. After the booting process is completed we will be in the command level of our Pascal system.

   1. To boot the Apple do the following:

      -<Place APPLE1: into #4: and close door>
      -<Place APPLE2: into #5: and close door>
      -<Turn on CRT>
      -<Turn on computer>

   2. Now wait and let the Apple do the rest. First, the message APPLE II will appear at the top of the CRT and disk drive #4: will start working. Then, there will be more flashing on the screen and more sounds from the disk drives. Finally, a message similar to this will appear, but with a different date:

```
WELCOME APPLE1, TO
U.C.S.D. PASCAL SYSTEM II.1
CURRENT DATE IS 26-JUL-79
```

3. A second or two later, the following line will appear at the top of the screen:

```
Command: E(DIT, R(UN, F(ILE, C(OMP, L(IN
```

This is the command level prompt line that lets you know that you are ready to run Pascal. Only APPLE1: is needed to boot, but it is better to have a diskette, any diskette, in #5: while booting.

4. The command level prompt line gives a list of most of the common commands available. These commands will be discussed briefly now, and then more fully in subsequent procedures.

   a. The Edit command is used to create or modify files of text. This file of text can be a Pascal program, a data file, or any other file of characters.

   b. The Compile command is used to translate the text of a Pascal program into the compressed P-code version of the program. This P-code is the machine language of the UCSD Pascal interpreter.

   c. The Execute command is used to execute a P-code file.

   d. The Run command is used to compile a text file and then execute the P-code file produced by the compiler.

   e. The File command is used to perform various file maintenance routines.

Formatting and Making Backup Copies

. In order to use diskettes for Pascal they must be properly formatted. This is similar to the initialization of diskettes for BASIC. In this section we will go through the steps necessary to format a diskette. Once a diskette is formatted it may be used to store programs or to make a backup copy of a Pascal system diskette. You should make a backup copy of each Pascal system diskette and use these diskettes instead of the original diskettes when you run Pascal. This will protect your original diskettes from destruction in the event of a system malfunction.

A. Formatting Diskettes

To format a diskette:

-<Boot Pascal system>
-<Remove APPLE2: from #5:>
-<Place APPLE3: into #5: and close the door> [We are making this switch because the Formatter program is on APPLE3:]
- Enter: X [for Execute command] Screen will display: EXECUTE WHAT FILE?
- Enter: APPLE3: FORMATTER<cr> After a few seconds, the screen will display:

```
APPLE DISK FORMATTER PROGRAM
FORMAT WHICH DISK (4,5,9..12) ?
```

- <Remove APPLE3: from #5:>
- <Place diskette to be formatted into #5:>
- Enter: 5 <cr> [indicating #5: as drive] Screen will display: NOW FORMATTING DISKETTE IN DRIVE 5 [It will take about 30 seconds to format the diskette.] Screen will then display: FORMAT WHICH DISK (4,5,9..12)?

[Warning: Any diskette placed in #5: will be formatted and any information on the diskette will be lost. The Formatter program will give you a warning if the diskette to be formatted is already formatted. For example, if you leave APPLE3: in #5: by mistake you will see the warning message: DESTROY DIRECTORY OF APPLE3? At this point, you can enter N and the diskette will not be destroyed. No warning message will be given if the diskette to be formatted is not already formatted.]

You may now format another diskette by placing it in #5: and entering 5 <cr>. When you are finished formatting diskettes:

- Enter: <cr > Screen will display: THAT's ALL FOLKS...

The command level prompt line will now appear across the top of the screen

B. Making Backup Copies of Diskettes

- <Get command level prompt line> [either from previous formatting procedure or by booting system]
- Enter: F [for Filer] Screen will display: FILER:G,S,N,L,R,C,T,D,Q [1.1]
- Enter: T [for Transfer] Screen will display: TRANSFER ?
- <Place diskette to be copied into one of the disk drives> [e.g., put APPLE3: into #4:]
- <Place one of the newly formatted diskettes into the other drive>
- Enter: APPLE3: <cr > Screen will display: TO WHERE?
- Enter: BLANK: <cr> [The Formatter program gives the name BLANK: to all newly formatted diskettes.] Screen will display: TRANSFER 280 BLOCKS? (Y/N)
- Enter: Y [To continue process] Screen will display: DESTROY BLANK: ?
- Enter: Y [to continue process] After about 2 minutes of copying and checking, the screen will display: APPLE3: -- BLANK: [This message is telling you that the contents of APPLE3:, including the diskette's name, have been copied on to the diskette that used to be called BLANK:. There are now two diskettes named APPLE3: in the disk drives. To avoid confusion, immediately remove the newly created APPLE3: and label it accordingly. At this point you are still in the Filer. You may enter T [for Transfer] to make another copy if you have more diskettes to back up.]

Once you have finished making all of your backup copies:

- <Place APPLE1: in #4:>
- Enter: Q [to Quit]

You will then see the command level prompt line displayed across the top of the screen.

Simple Filer Commands

. In this section we will go over some of the more frequently used Filer commands. Each of these commands is used while in the Filer with the Filer prompt line displayed as follows:

```
FILER:G,S,N,L,R,C,T,D,Q
```

. You get into Filer by entering F while the command level prompt line is displayed.

. Additional filer commands appear if the "?" key is pressed, including the "V" and "W" commands.

A. Date

This command allows you to change the date. All newly created files will contain this date. It is advisable to keep the date current.

- Enter: D [for Date] The computer will display the following lines:

```
DATE SET: <1..31>-<JAN..DEC>-<00..99>
TODAY IS 19-AUG-79
NEW DATE ?
```

-<Enter new date or just <cr> for no change>. [e.g., 30-May-84 <cr>]

B. Volumes

This command will list the input and output volumes (devices or diskettes) currently on-line (actively connected into the system), by volume name and by volume number.

- Enter: V [for Volumes] A typical display for a two-drive system is:

```
VOLS ON-LINE:
  1   CONSOLE:
  2   SYSTERM:
  4 # APPLE1:
  5 # APPLE2:
ROOT VOL IS - APPLE1:
PREFIX IS   - APPLE1:
```

The volumes CONSOLE: and SYSTERM: are always available. They are just two different ways to refer to the screen and keyboard. Block-structured devices (disks) are indicated by a pound sign (#). The boot diskette, also called the system volume, is indicated here as the ROOT VOL--in this case the diskette named APPLE1:. The default volume is indicated here as the PREFIX volume. In general, the prefix volume will be the same as the boot volume unless it has been changed.

C. List Directory

This command will list a diskette's directory. All files and unused areas are listed along with their block length and last modification date.

- Enter: L [for List Directory] Computer will display: DIR LISTING OF?
- Enter: #4:<cr> [or #5:<cr>]

With APPLE1: in #4: a typical response will contain most of the following files:

SYSTEM.APPLE, SYSTEM.PASCAL, SYSTEM.MISCINFO, SYSTEM.EDITOR, SYSTEM.FILER, SYSTEM.LIBRARY, SYSTEM.CHARSET, and SYSTEM.SYNTAX.

These files, along with those on APPLE2: and APPLE3:, are needed to use the various features of the Pascal operating system and compiler. As we create files, they too will appear on the directory of the diskette. By using the List Directory command we will be able to see which files our diskette contains.

D. Quit

This command will allow us to leave the Filer and return to command level.

- Enter: Q [for Quit]

Editor

. In this section we will see how to create and modify a text file using some of the basic Editor commands. When we use the Editor the following special keys and commands are often needed:

<CTL-K>	The APPLE II+ keyboard does not have brackets, but this command will produce a left bracket (i.e., [).
<SHIFT-M>	This is for a right bracket (i.e.,]).
	The right arrow key is used to move the cursor to the right.
	Used to move the cursor to the left.
<CTL-O>	This will move the cursor up one line.
<CTL-L>	This will move the cursor down one line.
<CTL-A>	The Apple Pascal Editor is set up for a width of 80 characters, but without an 80-column card in the Apple you will see only the leftmost 40 characters. This command will shift your view to the rightmost 40 characters. Pressing <CTL-A> again will shift the view back to the leftmost 40 characters.

<CTL-Z>   This will put the Apple in the auto-follow mode. Your viewing screen will shift to the right or left to follow the cursor. This is very handy when you are entering many long lines of text. <CTL-A> and many other commands will cancel the auto-follow mode.

. When we use the Editor to create and save a file, the file will be saved under the name SYSTEM.WRK.TEXT on the APPLE1: diskette. Before we begin using the Editor we will briefly return to the Filer and use the New command to clear the workfile for our use. [If the diskette you are using belongs to someone else, check with them before you clear the workfile. This process will cause their workfile to be lost in the process.]

- <Get command level prompt line>
- Enter: F [for Filer]
- Enter: N [for New] [If there was a previous workfile, you will get this prompt: THROW AWAY CURRENT WORKFILE? If you do, respond by entering Y to dispose of the old workfile.] Computer will display: WORKFILE CLEARED
- Enter: Q [to Quit Filer]
- Enter: E [for Editor] Computer will display the following message: NO WORKFILE IS PRESENT. FILE? (<RET> FOR NO FILE <ESC-RET> TO EXIT)
- Enter: <cr> [to start a new file] Computer will display the following:

```
>EDIT: A(DJST C(PY D(LETE F(IND I(NSRT J(MP R(PLACE Q(UIT X(CHNG Z(AP [1.1
```

- Enter: I [for Insert] Computer will display the following:

```
>INSERT: TEXT [<BS> A CHAR, <DEL> A LINE] [<ETX> ACCEPTS, <ESC> ESCAPES]
```

. With the insert prompt line across the top of the screen you are in the insert mode. You may now insert text (i.e., characters) into the text file. When text is entered into the system, we will refer to three different places: workfile, workspace, and buffer. The following diagram will be referred to as we begin to use the Editor. Do not worry if the diagram is not clear to you now. The workfile is the file that is on the diskette. This is the only file that is saved when the computer is turned off. It will usually be the file SYSTEM.WRK.TEXT on APPLE1:. The workspace is the file that is currently in the computer's memory. It contains all the additions and modifications you make to your file while using the Editor. The buffer contains additions and modifications that are currently in progress, but which have not yet been made to the workspace.

WORKFILE
(on Diskette)

Quit Editor Update   Enter Editor

WORKSPACE
(In Computer)   Quit-Editor Exit without Update or turn off Power   LOST

leave Insert by <CTL-C>

BUFFER
(In Computer)   ESC from Insert or turn off power   LOST

enter Insert type text

. If you are not in the insert mode, follow the steps listed previously to get the insert mode. Enter the following data <u>exactly</u> as it appears, including spacing. We have intentionally included errors so that we can illustrate the Editor commands that will be used to modify your text files.

- Enter:

```
ONE FOUR THE MONYE <CR>
EXTRA LINE <cr>
  TWO FOR THE SHOW <cr>
AND FOR TO GO <cr>
```

- Enter: <CTL-C> [As you were typing the above text it was being entered into the buffer area. <CTL-C> signifies that you are finished inserting text and that you want this new text inserted into the workspace. If you enter <ESC>, the new text will be lost and the workspace will not be changed.]

- Enter: Q [to Quit the Editor] The following lines will be displayed:

```
    QUIT:
         U(PDATE THE WORKFILE AND LEAVE
         E(XIT WITHOUT UPDATING
         R(ETURN TO THE EDITOR, WITHOUT UPDATING
         W(RITE TO A FILE NAME AND RETURN
         S(AVE WITH SAME NAME AND RETURN
```

- Enter: U [to Update the workfile and leave. This will copy the workspace to the workfile on the diskette. If you had a workfile on the diskette it would be replaced by the current file in the workspace in the computer's memory. Any additions and changes made to the text file would now be saved on the diskette. We will verify this in a moment.] Computer will now display the command prompt line [since we left the Editor]
- <Turn off computer>
- <Wait a minute and turn on computer>
- Enter: F [for Filer]
- Enter: L [for List Directory]
- Enter: #4:<cr> [for APPLE1:] Among the files listed you should see our newly created file with the name SYSTEM.WRK.TEXT.
- Enter: Q [to Quit Filer]
- Enter: E [for Editor. Since we now have a workfile present on the diskette, it will automatically be copied into the computer's workspace when we enter the Editor. You should now see the text previously entered. We will now use the Insert, Delete, Exchange, and Adjust commands of the Editor to correct our text file. When you are asked to move the cursor to a certain location on the screen you will use the following keys:<→>(right),<←>(left),<CTL-O>(up), and <CTL-L>(down).]
- <Move cursor to the Y on the first line>
- Enter: X [for Exchange command]
- Enter: EY [to exchange EY for YE]
- Enter: < CTL-C > [to complete the exchange and to change the workspace]
- <Move cursor to the first letter of the third line (T)>
- Enter: A [for Adjust command]
- Enter: ← ← [The third line of text did not line up with the other lines of our text. By using the Adjust command with the ← or → keys we can move the entire line to the left or right.]
- Enter: <CTL-C> [to complete the Adjust command and change the workspace]
- <Move cursor to R on last line of text>
- Enter: I [for Insert] R and all characters to the right of R will move over to the right of the screen (possibly completely off the screen).
- Enter: U [to correct spelling of FOUR]
- Enter: < CTL-C> [to insert into workspace] The text on the right of the screen will have now been shifted back to the left and the U is inserted.
- <Move cursor to the space immediately to the right of the last letter of the third line (W).>
- Enter: I [for Insert; to insert new line]
- Enter: <cr> THREE TO GET READY [to insert new line]
- Enter: <CTL-C> [to insert into workspace]

- <Move cursor to U on first line>
- Enter: D [for Delete]
- Enter: <spacebar> [to delete the U]
- Enter: <CTL-C> [to delete from workspace. The U is now gone and the word FOR is now correct. Note that the Delete command removed the letter U and the space that it was taking up.]
- <Move cursor to the first letter of the second line (E)>
- Enter: D [for Delete; to delete second line]
- Enter: <cr> [to delete entire line]
- Enter: <CTL-C> [to delete from workspace] Our text file should now appear as:

ONE FOR THE MONEY
TWO FOR THE SHOW
THREE TO GET READY
AND FOUR TO GO

If you don't have this now, make whatever changes are necessary. You may want to make other changes at this time to get more practice using the various Editor commands. Just remember that the changes made so far are only in the computer's workspace. The workfile on the diskette contains the original file with all of the errors.

- Enter: Q [to Quit Editor]
- Enter: U [to update the workfile on the diskette, i.e., SYSTEM.WRK.TEXT]

- By rebooting and entering the Editor you can verify that the last update command we gave did in fact save all of our changes onto the workfile. Our original workfile, with the errors, is now lost and replaced with the corrected file.

- Before we proceed to write a Pascal program, we will go over some of the other options given to us when we quit the Editor. Up to now we have only used the Update option. The Exit option has us exit the Editor, but our workfile on the diskette does not get updated. All changes made since the last update are lost forever. The Return option will return us to the editor but will not update the workfile on the diskette. The Write option will allow us to copy the workspace onto a file other than SYSTEM.WRK.TEXT. If you choose this option, you will be asked to give the name of the file (and diskette, if other than APPLE1:). A suffix of text will be added to the file name and APPLE1: will be chosen if you do not specify a diskette. Therefore, a response of MYFILE<cr> will save the workspace as APPLE1:MYFILE.TEXT. You will then be returned to the Editor and will have to quit again, using either the Update or Exit option, depending on whether or not you want to update the usual workfile, SYSTEM.WRK.TEXT. The Save option will be gone over later when we create a Pascal text file.

Writing and Running a Pascal Program

A. Now that we know how to use the Editor to write text, let us write an actual Pascal program we can then execute.

1. Suppose we are at the command level.

```
- Enter:  F <cr>  [to enter Filer]
- Enter:  N <cr>  [to clear workfile SYSTEM.WRK.TEXT]
- Enter:  Q <cr>  [to Quit Filer]
- Enter:  E <cr>  [to Enter Editor]
- Enter:  <cr>  [to indicate we want an empty workspace]
- Enter: I  [to insert text of our program]
- Enter:

      PROGRAM FIRST (OUTPUT);
      BEGIN
        WRITELN ('THIS IS OUR FIRST PROGRAM.')
        WRITELN ('I HOPE IT WORKS.')
      END.
- Enter: <CTL-C>
- Enter:  Q U C  [to Quit the Editor, Update (or create) workfile
  SYSTEM.WRK.TEXT and Compile the program]
```

2. The compilation will not work because we left out a semicolon at the end of our first writeln statement, and we will get the following output.

```
COMPILING....

PASCAL COMPILER
<  0>..
FIRST
<  2>
  WRITELN ('THIS IS OUR FIRST PROGRAM.')
  WRITELN <<<<
  ILLEGAL SYMBOL (TERMINATOR EXPECTED)
  LINE 3, ERROR 6:<SP>(CONTINUE), <ESC>(TERMINATE), E(DIT
```

We have three options:

- Enter: <space bar> [to continue compilation. Though this may help locate other errors, it is often inadvisable, since one error can cause nonexistent errors to appear on the screen.]
- Enter: <ESC> [to end compilation and return to the command level]
- Enter: E [to enter the Editor and position the cursor next to the error on the screen]

2. Select the third option and correct the error on the screen:

   - Enter: E
   - Press: <spacebar> [to get the edit commands on the top of the screen.]

3. We can now fix our error by inserting a semicolon. It will then be necessary to recompile our program:

   - <Move cursor to the space immediately to the right of the last letter of the third line()).>
   - Enter: I [for Insert]
   - Enter: ; [to insert;]
   - Enter: <CTL-C> [to insert into workspace]
   - Enter: Q U C [to Quit the Editor, Update workfile, and Compile the program]

4. Our program will now compile, and a file SYSTEM.WRK.CODE will appear on the boot diskette. (Check this by going to the filer and listing the boot directory, APPLE1:.) The compilation process translates the source code (SYSTEM.WRK.TEXT) into object code (SYSTEM.WRK.CODE) which can now be executed.

5. We can now run the program:

   - Enter: R [to Run the program]

   The screen now displays the text:

   ```
   RUNNING...
   THIS IS OUR FIRST PROGRAM.
   I HOPE IT WORKS.
   ```

6. Let's run the program again:

   - Enter: R

   The program now runs more quickly than before, since SYSTEM.WRK.CODE already exists on the diskette. We could have also used X(ecute to run the program. The difference between R(un and X(ecute is that X(ecute executes a code file while R(un both compiles the text file (if necessary) and executes the code file that is produced.

B. Suppose we now decide to write another program while retaining the first program under the name Message.

1. This can be done by going to the filer and using the S(ave command:

   - Enter: F
   - Enter: S [to Save the workfile]

   When the question, SAVE AS ? appears:

   - Enter: MESSAGE<cr>

   SYSTEM.WRK.TEXT and SYSTEM.WRK.CODE have now been renamed MESSAGE.TEXT and MESSAGE.CODE. Use the List command to check this in the directory. These two newly named files are now designated as the workfile even though the SYSTEM.WRK.FILE names have been eliminated. We can check that MESSAGE.TEXT and MESSAGE.CODE are designated the workfile by using the What command in the filer, which tells us the current status of the workfile. The What command is executed by entering W while you are in the Filer. If we try to save a file with a name that is identical to one already on the diskette, the older one will be erased.

2. Now that we have saved our original workfile under the name Message, we can clear the workfile using New and write a new program:

   - Enter: N [for New]
   - Enter: Q [to Quit the Filer]
   - Enter: E [to Enter the Editor]

3. As expected, no workfile is present:

   - Enter: <cr> [to indicate we want an empty workspace]
   - Enter: I [to Insert our next program called Second]
   - Enter:

```
PROGRAM SECOND (OUTPUT);
VAR
  A, B, SUM : INTEGER;
BEGIN
  A := 5;
  B := 7;
  WRITELN ('THE SUM OF', A : 3, ' AND', B : 3, ' IS', A + B : 3)
END.
```

   - Enter: <CTL-C> [to complete insertion]
   - Enter: Q U R [to Quit, Update, and Run]

4. If there are no errors, the screen will display:

```
RUNNING...
THE SUM OF  5 and  7 IS 12
```

5. Suppose we now decide to save the new program under the file name Second and retrieve program First (which is stored as file name Message) to make changes:

   - Enter: F S [to enter Filer and Save]

   When the prompt SAVE AS ? is displayed:

   - Enter: SECOND<cr>
   - Enter: N [to clear workfile)
   - Enter: G [Get command designates a file to be used as the workfile. Though the file name is not changed, it is marked as the workfile]

   When the prompt GET ? is displayed:

   - Enter: MESSAGE [The message TEXT AND CODE FILE LOADED will appear on the screen. MESSAGE.TEXT and MESSAGE.CODE are now designated as the workfiles. This can be verified by using the What command.]

6. The editor can now be called to make changes in program First (in file Message) and Update. Update causes SYSTEM.WRK.TEXT to be created.

Notes:

. When students start using Save and Get they often have difficulty understanding what the system designates as the workfile and the change in filenames on the diskette. It is suggested that they use the What and List commands in the filer after each command until they know how the commands operate.

. If we use Get to designate a workfile, Edit the file, and then Update, the system will have the original file designated as the workfile and the updated version, SYSTEM.WRK.TEXT. By calling Save, we have the option of renaming SYSTEM.WRK.TEXT to the designated workfile or saving SYSTEM.WRK.TEXT under a new user-defined name. In this way Get and Save can be used to make modifications in previously written programs.

. Another way to save the workspace as a file on a diskette is to use Write when quitting the Editor. This is especially useful when there is no more room on the boot diskette.

. Beginners may be confused by the messages displayed by the What command:

MESSAGE	WORKFILE STATUS
NO WORKFILE	No SYSTEM.WRK.TEXT, SYSTEM.WORK.CODE, or designated file
WORKFILE IS DISKNAME:FILENAME	Though SYSTEM.WRK.TEXT and SYSTEM.WRK.CODE do not exist on the boot diskette, some diskette contains a designated workfile
NOT NAMED (NOT SAVED)	SYSTEM.WRK.TEXT or SYSTEM.WRK.CODE exists on boot diskette and no other file is designated as the workfile
WORKFILE IS DISKNAME:FILENAME (NOT SAVED)	SYSTEM.WRK.TEXT or SYSTEM.WRK.CODE exist on boot diskette; this is the most recent copy of the designated workfile on the boot diskette

Copying Diskettes

The Transfer command is used to copy a file or entire diskette to another volume

A. To copy an entire diskette:

- Enter: T
  Prompt: TRANSFER ?
- Enter: <diskname1:>
  Prompt: TO WHERE ?
- Enter: <diskname 2:>
  Prompt: TRANSFER 280 BLOCKS? (For IBM: 320 blocks)
- Enter: Y
  Prompt: DESTROY DISKNAME2: ?
- Enter: Y

The message DISKNAME1: -> DISKNAME2: confirms the transfer is completed.

B. To copy a file from one diskette to another:

- Enter: T
  Prompt: TRANSFER ?
- Enter: DISKNAME1: FILENAME1
  Prompt: TO WHERE ?
  Enter: DISKNAME2:FILENAME 2

The message DISKNAME1:FILENAME1 -> DISKNAME2:FILENAME2 confirms the transfer is complete.

Notes:

. #4: and #5: may be used in place of disknames.
. No diskname is needed for the boot volume.
. The disknames or filenames may be the same.

C. To copy a file from a diskette to the printer:

Follow the directions in (B) above, but enter PRINTER: or #6: in place of DISKNAME2:FILENAME2

Using Lesser Filer and Editor Commands

This is a brief introduction to additional user Filer and Editor commands that may be introduced as the term progresses:

A. Using additional Filer commands

- R(EMOVE: Erases a file name from its diskette directory. The user only has to indicate the file to be erased when the prompt REMOVE WHAT FILE? is displayed.

- C(HANGE: Renames a file or diskette.

- K(RUNCH: Crunches all files on a diskette so that all unused areas are consolidated. This command is used when one is about to run out of space because the unused areas on the diskette are fragmented.

- B(AD BLOCKS: Identifies bad blocks on the diskette.

- X(AMINE: Tries to repair bad blocks.

Bad blocks may be due to damage on the diskette or poor system connections. The Bad Blocks command should be used when one has an unexplainable error. Xamine tries to fix bad blocks. If it fails, the user can mark these block as bad; files containing these blocks are removed from the directory.

B. Using additional Editor commands

- P(AGE: Moves the cursor one page.

- J(UMP: Moves the cursor to the files BEGINNING, END, or MARKER. A marker may be set in a file by using the Editor's Set command. Place the cursor at the location where the marker is to be set and then enter S. (Note: The Set command can still be used even though it may not appear as one of the listed options). Press the Marker option and give the marker a name that can either be a number or identifier. It is useful to set markers when we want to jump to different parts of the program quickly.

- C(opy: Inserts buffer or file text at cursor position. Text is put in a copy buffer after we leave the insert or delete mode. The copy command can be used to move text from one position to another in the workspace by deleting the text, placing the cursor at the new position and using Copy Buffer. To make duplicate copies of text without erasing, follow the technique just described but press <ESC> instead of <CTL-C>. Students can combine different parts of a program that they have on separate files by using Copy File (markers can be used).

- F(IND: Moves the cursor to a specified occurrence of a given string. Place the cursor where you want to begin the search and enter F. When a target is requested, enter: /<target string>/. The cursor will appear at the end of the first occurrence it finds of the <target string>. Press a number (N) or slash (/) before F and then FIND [N] or FIND [/] will appear on the screen and the Nth or last occurrence of the <target string> is sought. (Note: Any other pair of delimiters may be used in place of "/" to enclose the <target string>. Isolated occurrences of the <target string> (tokens) are searched for unless the Literal option (L) is used. These isolated occurences of the tokens are copies of the <target string> preceded and followed by any character that is not a letter or number. With the Literal option (L), the <target string> is searched for, even if it is embedded within a word.)

- R(EPLACE: Replaces a specified number (N) of occurrences of a string with a substitute string. This command is similar to Find. To replace the first five occurrences of the token NUM by NUMBER, enter: 5R. When REPLACE [5] appears, type /NUM/ /NUMBER/. All occurrences can be replaced by typing a slash (/) before R(EPLACE.

APPENDIX G

SCOPE & SEQUENCE

Advanced Placement Computer Science with Pascal

The following topics and performance objectives encompass the lessons in "Advanced Placement Computer Science - Pascal Volume I." This course is recommended for students in grades 11 and 12.

VOLUME I TOPICS

Topic 1 - Computer Components

Performance Objectives - Students will be able to:

- Identify the four major components of the computer; CPU, memory (main and secondary), input devices, output devices.
- Explain the functions of each of the major components.
- Differentiate between RAM and ROM.

Topic 2 - Computer Languages

Performance Objectives - Students will be able to:

- State the need for a variety of computer languages.
- Describe the characteristics of some of the more popular languages.
- Differentiate Pascal from BASIC.

Topic 3 - Compilers

Performance Objectives - Students will be able to:

- Explain the difference between a compiler and an interpreter.
- Describe the compilation process.
- Explain the difference between compile time, run time, and output errors.
- Identify the processing stage where errors are detected.

Topic 4 - Essential Features of a Pascal Program

Performance Objectives - Students will be able to:

- Recognize essential features of all Pascal programs (program heading, BEGIN and END).
- State the difference between reserved words and identifiers.
- Use a simple write or writeln statement in a program.

Topic 5 - Pascal Variables

Performance Objectives - Students will be able to:

- Give the name, type, and value of a variable.
- Input information interactively using the read and readln statements with prompt messages.
- Write simple variable declarations and assignment statements.

Topic 6 - Data Types and Constants

Performance Objectives - Students will be able to:

- List and describe the four predefined data types.
- State the difference between constants, variables, and literals.
- Explain the benefits of using constant declarations.
- Write a constant declaration.
- Find the largest and smallest possible integer values that can be used for a given computer.

Topic 7 - Real and Integer Data Types

Performance Objectives - Students will be able to:

- Evaluate integer and real expressions using the precedence rules for operators.
- Explain the terms range, precision, and exponential notation for real numbers.
- State when implicit type conversion occurs between integers and reals.
- Indicate how the writeln statement may be used with attributes.
- Discuss the limitation of precision for reals, and its consequences.

Topic 8 - Character and Boolean Data Types

Performance Objectives - Students will be able to:

- Give examples of appropriate values for each data type.
- Explain what is meant by ASCII.
- Evaluate boolean and relational expressions.
- Explain what flags are.

Topic 9 - Standard or Built-In Functions

Performance Objectives - Students will be able to:

- List the standard functions.
- Indicate the argument type and result type for each standard function.
- Write expressions and statements using standard functions.
- Write an expression to convert degree measures to radian measures and vice versa.
- Write an expression to round off (truncate) a real number to a given number of decimal places.
- Use standard functions to evaluate other commonly used mathematical functions, such as exponentiation, $\log_{10}x$, 10^x, tan (x), arcsin (x), ln (x).

Topic 10 - If-Then Statements

Performance Objectives - Students will be able to:

- Explain how the IF-THEN statement is used by a computer to make run-time decisions.
- Describe how compound statements can be used in an IF-THEN statement.
- Diagram (flowchart) the sequence of actions in an IF-THEN statement.

Topic 11 - If-Then-Else Control Structures

Performance Objectives - Students will be able to:

- Explain and diagram the actions of an IF-THEN-ELSE statement.
- Use the IF-THEN-ELSE statement in programs that involve decisions between two alternative courses of action.
- Show how a value can be assigned to a boolean variable with an arithmetic assignment statement and then use this boolean variable in an IF-THEN-ELSE statement to make the coding read very much like English.

Topic 12 - Nested If-Then-Else Statements

Performance Objectives - Students will be able to:

- Define a compound IF statement (nested conditional statement).
- Diagram compound IF statements from program code and write program code from diagrams.
- State the rule by which the computer deals with the apparent ambiguity of the dangling ELSE clause.

Topic 13 - Programs Involving If-Then-Else Statements

Performance Objectives - Students will be able to:

. Use IF-THEN-ELSE statements in computer programs.
. Identify problems that arise through improper use of compound statements.

Topic 14 - Repeat-Until Structures

Performance Objectives - Students will be able to:

. Use the REPEAT-UNTIL statement.
. Define a delimiter and give examples in Pascal.

Topic 15 - While-Do Statements

Performance Objectives - Students will be able to:

. Explain the syntax and operation of the WHILE-DO statement.
. Use the WHILE-DO statement in computer programs.
. State the purpose of a sentinel value.
. State the difference between the WHILE-DO and REPEAT-UNTIL statements.

Topic 16 - For-Do Statements

Performance Objectives - Students will be able to:

. Explain the syntax and operation of the FOR-DO statement.
. Use the FOR-DO statement in computer programs.
. Identify the difference between the FOR-DO statement and other control statements.
. Use nested loops in writing programs.

Topic 17 - Case Statements

Performance Objectives - Students will be able to:

. Explain the syntax and operation of the CASE statement.
. Use the CASE statement in computer programs.
. Cite examples of situations in which the CASE statement can be used effectively.

Topic 18 - Standard Files: Input and Output

Performance Objectives - Students will be able to:

- Describe the standard input and output files.
- Insert statements in a program to direct output to the printer.
- Format output neatly using the write and writeln procedures.

Topic 19 - EOF and EOLN Functions

Performance Objectives - Students will be able to:

- State the purpose of the EOF and EOLN functions.
- Indicate when the EOF and EOLN functions take on the values true and false.

Topic 20 - Stepwise Refinement

Performance Objectives - Students will be able to:

- Describe a systematic technique for solving problems on the computer.
- Identify the essential ideas in stepwise refinement.
- Analyze a problem using stepwise refinement.

Topic 21 - Inventory Management

Performance Objectives - Students will be able to:

- Explain the process by which a company takes inventory.
- Analyze a major computer business application using stepwise refinement.

Topic 22 - Functions

Performance Objectives - Students will be able to:

- Explain the syntax of functions.
- Describe the operation of functions.
- Differentiate between formal and actual parameters.
- Describe a local variable.
- Discuss how functions interact with the main program.

Topic 23 - Local Variables and Side Effects

Performance Objectives - Students will be able to:

- Tell the difference between local and global variables.
- Explain how side effects can be avoided in functions.
- Explain how functions can be nested.

Topic 24 - Procedures Without Parameters

Performance Objectives - Students will be able to:

- Describe the syntax of procedures without parameters.
- Explain how procedures interact with the main program.
- State the reasons for creating procedures.
- Explain the difference between a function and a procedure.
- Write a procedure without parameters.

Topic 25 - Procedures With Parameters

Performance Objectives - Students will be able to:

- Write procedures with parameters.
- Explain the difference between actual and formal parameters.
- Explain the difference between value and variable parameters.
- Contrast the effects of transmitting information to a procedure by value and by variable parameters.

Topic 26 - Parameters in Procedures and Functions

Performance Objectives - Students will be able to:

- Explain when a parameter should be passed by value or passed by reference.
- List reasons for using parameters in procedures and functions.

Topic 27 - Scope of an Identifier

Performance Objectives - Students will be able to:

- Define the scope of an identifier.
- Describe the scope of identifiers in subprograms.
- Explain the relationships between the scope of an identifier, side-effects, and parameters.
- Contrast domain of accessibility and existence.
- Write nested procedures and functions.

Topic 28 - Forward References

Performance Objectives - Students will be able to:

- State the purpose of forward references.
- Describe how forward references are implemented.
- Explain how functions and procedures can be used as parameters.

Topic 29 - Top-Down Design

Performance Objectives - Students will be able to:

- State the advantages of top-down design.
- Contrast the top-down and bottom-up approaches to problem solving.
- Define program modularization and discuss its benefits.

Topic 30 - Programming Methodology

Performance Objectives - Students will be able to:

- List the essential elements of good programming methodology.
- Discuss the concepts of specification, design, coding, correctness, and documentation.

Topic 31 - Program Correctness

Performance Objectives - Students will be able to:

- Differentiate between several levels of program correctness.
- Discuss the process of program testing.
- Describe various types of errors and diagnostic techniques.

Topic 32 - Recursive Functions: Part I

Performance Objectives - Students will be able to:

- Define a recursive function.
- Explain how recursion works for finding factorials.
- Discuss when recursion can be used in problem solving.

Topic 33 - Recursive Functions: Part II

Performance Objectives - Students will be able to:

- Explain how recursion works for finding Fibonacci numbers.
- Discuss the advantages and disadvantages of recursion.
- Write simple recursive functions.

Topic 34 - Indirect Recursion

Performance Objectives - Students will be able to:

- Define indirect recursion.
- Explain how forward references are used with indirect recursion.
- Describe how a recursive function stacks data items.

Topic 35 - Towers of Hanoi Problem

Performance Objectives - Students will be able to:

- State the Towers of Hanoi problem.
- Show how the problem leads to a recrusive solution.
- Provide the recursive calls from any given state in a solution.

Topic 36 - Iteration vs. Recursion

Performance Objectives - Students will be able to:

- Define several classes of algorithms.
- Explain the difference between iterative algorithms and recursive algorithms.
- Write recursive functions iteratively and iterative functions recursively.

Topic 37 - User-Defined Scalar Types

Performance Objectives - Students will be able to:

- Create user-defined scalar types.
- Explain how user-defined scalar types are used in control and assignment statements.
- Apply certain functions to user-defined scalars.
- List major limitations of user-defined scalars.

Topic 38 - Subrange Types

Performance Objectives - Students will be able to:

- Write subrange type declarations.
- State the reasons for using subranges.

SCOPE & SEQUENCE

Advanced Placement Computer Science with Pascal

The following topics and performance objectives encompass the lessons in "Advanced Placement Computer Science - Pascal Volume II." This course is recommended for students in grades 11 and 12.

VOLUME II TOPICS

Topic 39 - Arrays, a Structured Data Type

Performance Objectives - Students will be able to:

- Explain why structured types are needed.
- Define an array and related terms.
- Write an array type declaration.

Topic 40 - Using Arrays In Programs

Performance Objectives: Students will be able to:

- Use arrays to solve programming problems.
- Explain when arrays are used in problem solving.

Topic 41 - Arrays as Parameters in Procedures

Performance Objectives - Students will be able to:

- Explain how to manipulate arrays when the exact number of elements is determined during execution.
- Write procedures using arrays as parameters.
- Indicate why an array is a random access structure.

Topic 42 - Sorting

Performance Objectives - Students will be able to:

- Discuss the importance of sorting arrays.
- Describe bubble sort techniques.
- Write a bubble sort procedure.

Topic 43 - Binary Search

Performance Objectives - Students will be able to:

- State the need for efficient search techniques.
- Identify problems related to a sequential search.
- Explain how the binary search method reduces search time.
- Write a recursive binary search function.

Topic 44 - Implementing Multi-Dimensional Arrays

Performance Objectives - Students will be able to:

. Declare multi-dimensional arrays.
. Show input and output operations on two-dimensional arrays.
. Explain how calculations can be performed on rows and columns of two-dimensional arrays.

Topic 45 - Packing Array Elements to Reduce Storage

Performance Objectives - Students will be able to:

. Explain how the usual implementation of arrays wastes storage space.
. Describe the techniques for packing arrays.
. Discuss the trade off between packing and not packing arrays.
. State the restriction on using packed arrays.

Topic 46 - Implementing Strings with Packed Arrays

Performance Objectives - Students will be able to:

. Explain how strings are implemented using packed arrays.
. Discuss the advantages of defining a string as a packed array of char.
. Show how to simulate the printing of user-defined scalars.

Topic 47 - Implementing Strings in UCSD Pascal

Performance Objectives - Students will be able to:

. Discuss differences between the UCSD and standard implementation of strings.
. Indicate how UCSD string functions and procedures work.

Topic 48 - UCSD String Processing Applications

Performance Objectives - Students will be able to:

. Describe how (UCSD) string functions can be used to decipher codes.
. Write meaningful programs using (UCSD) string functions.
. Discuss several applications of string processing.

Topic 49 - Standard Pascal String Processing

Performance Objectives - Students will be able to:

. Describe the difficulties in implementing strings in standard Pascal.
. Discuss methods for implementing strings in standard Pascal.
. Write string functions and procedures for standard Pascal.

Topic 50 - Sets in Pascal and Their Operations

Performance Objectives - Students will be able to:

- Write set declarations.
- Define and implement set operations: union, intersection, difference and IN.
- Explain how the relational operators can be used to compare sets.

Topic 51 - Illustration of Set Properties

Performance Objectives - Students will be able to:

- Discuss the advantages and disadvantages of using sets.
- Explain how the elements in a set can be printed.
- Indicate how sets can be simulated using a boolean array.

Topic 52 - Pascal Records

Performance Objectives - Students will be able to:

- State how records differ from arrays.
- Write record type definitions.
- Declare arrays of records.

Topic 53 - Accessing Information Stored in Records

Performance Objectives - Students will be able to:

- Access data stored in a record using the "." notation and the WITH statement.
- Describe how records can be compared.

Topic 54 - Accessing Data Stored in Hierarchial Records

Performance Objectives - Students will be able to:

- Explain how hierarchial records can be accessed with nested "WITH" structures.
- Describe the need for record variants.
- Declare records that have fixed and/or variant parts.

Topic 55 - Accessing Data Stored in Variant Records

Performance Objectives - Students will be able to:

- Explain how the fields of a variant record can be accessed.
- Describe how case statements are used with variant records.

Topic 56 - Computer in Society

Performance Objectives - Students will be able to:

- Describe the tremendous impact computers are having on our society.
- Explain how to use and not abuse a computer system.
- Cite problems related to computers and individual rights of privacy.
- Discuss challenges to the security of computer systems.

Topic 57 - Introduction to Files

Performance Objectives - Students will be able to:

- Indicate the need for files.
- Tell the difference between sequential and random access files.
- Write a file type definition.

Topic 58 - Writing Files

Performance Objectives - Students will be able to:

- Describe how components are written to a file.
- Explain the purpose of the buffer variable.
- Show how a file of records is created.

Topic 59 - Reading Files

Performance Objectives - Students will be able to:

- Describe how components can be read from a file.
- Indicate how the buffer variable is used for every file access.
- Explain the relationship between put, get, read and write.
- Demonstrate how the fields of a file record can be accessed.

Topic 60 - Combining Files

Performance Objectives - Students will be able to:

- Append records to a file.
- Concatenate two files.
- Write a program that merges two files.

Topic 61 - Updating Files

Performance Objectives - Students will be able to:

- Describe the need to update files.
- Update a sequential file.
- Show how the UCSD SEEK procedure can be used to update a random access file.

Topic 62 - Textfiles

Performance Objectives - Students will be able to:

- . Describe how textfiles differ from other user-defined files.
- . Explain the special functions and procedures that apply to textfiles.
- . Write a program that copies a textfile.

Topic 63 - Textfile Applications

Performance Objectives - Students will be able to:

- . Find a specific record in a textfile which uses end of line markers to separate records.
- . List three ways textfiles can be used to improve data processing.
- . Explain how integer and real values are read from textfiles.
- . Discuss differences between the predefined input/output files and textfiles.

Topic 64 - Stacks - A Dynamic Data Structure

Performance Objectives - Students will be able to:

- . Explain the differences between dynamic and static data structures and the appropriateness of each.
- . Define the concept of a stack.
- . Explain how the size of a stack can dynamically change.
- . Explain the operations that may be applied to a stack.

Topic 65 - Implementing A Stack

Performance Objectives - Students will be able to:.

- . Describe how a stack may be implemented using the structure of a record.
- . Discuss the problems of stack underflow and overflow.
- . Write subprograms that implement the pop and push operations, and functions to determine if the stack is empty or full.

Topic 66 - Using A Stack For Error Checking

Performance Objectives - Students will be able to:

- . Describe how a stack can be used to verify whether parentheses are correctly nested in an expression.
- . Explain a pseudo-Pascal coded algorithm that implements the above check for proper nesting in an expression.

Topic 67 - Postfix Expressions

Performance Objectives - Students will be able to:

- Evaluate postfix and prefix expressions.
- Change expressions from postfix or prefix to infix form and vice-versa.
- Describe an algorithm that uses stacks to evaluate postfix expressions.

Topic 68 - The Structure of a Queue

Performance Objectives - Students will be able to:

- Define the concept of a queue.
- Explain operations that may be applied to queues.
- Discuss how a queue can be represented in Pascal.

Topic 69 - Implementation of Queue Operations

Performance Objectives - Students will be able to:

- Discuss problems related to programming various routines using the ring representation of a queue.
- Explain how the queue operations of remove, insert, empty, full, print and next can be implemented in Pascal.

Topic 70 - Linear Linked List

Performance Objectives - Students will be able to:

- Cite the major disadvantages of using arrays to represent stacks and queues.
- Define a linked list and related terms.
- Use algorithmic notation to demonstrate how elements are inserted or removed from a linked list.

Topic 71 - Stacks and Queues Using Linked Lists

Performance Objectives - Students will be able to:

- Describe the linked list representation of stacks.
- Describe the linked list representation of queues.
- Discuss how an available list can be used to implement the getnode and freenode operations.

Topic 72 - Pascal Implementation of Linked Lists

Performance Objectives - Students will be able to:

- Indicate the advantages and disadvantages of using lists in place of arrays.
- Show how a linked list can be implemented in Pascal using an array of records.
- Write Pascal subprograms that illustrate the freenode and getnode operations.

Topic 73 - Linked List Operations in Pascal

Performance Objectives - Students will be able to:

- Discuss problems related to the insertion and deletion of items in a linked list.
- Write Pascal procedures that insert and delete items in a linked list.
- Describe how an item can be placed in its proper position in a sorted linked list.

Topic 74 - Circular Linked List

Performance Objectives - Students will be able to:

- Define a circular linked list.
- Discuss operations on circular linked lists.
- State the Josephus Problem.
- Describe an algorithm that uses a circular linked list to solve the Josephus problem.

Topic 75 - Header Node and Doubly Linked List

Performace Objectives: Students will be able to:

- Define a header node.
- List several applications of header nodes.
- Discuss how a doubly linked list eliminates several drawbacks of a linear linked list.

Topic 76 - Pointer Variables

Performance Objectives - Students will be able to:

- Cite the disadvantages of using an array implementation for linked lists.
- Show how pointer variables can be declared in Pascal.
- Describe how the NEW and DISPOSE procedures can be used to dynamically allocate and free storage.
- Discuss the problem of using a dangling pointer.

Topic 77 - Dynamic Linked List

Performance Objectives - Students will be able to:

- Explain how pointer variables can be used to define nodes of a linked list.
- Write a procedure to dynamically create a linked list.
- Write a procedure to insert elements in a linked list.

Topic 78 - Linked List Operations

Performance Objectives - Students will be able to:

- Write a procedure to delete an element from a dynamic linked list.
- Write a procedure to insert and delete items at the end of a dynamic linked list.
- Discuss the advantages and disadvantages of the dynamic and array implementations of a linked list.

Topic 79-80 - Airline Reservation System

Performance Objectives - Students will be able to:

- Discuss major business applications of linked lists.
- Explain how top-down design can be effectively applied to a large-scale problem.
- Show how a linked list can be used to create an airline reservation system.
- Describe how a dynamically-implemented linked list can be maintained.

Topic 81 - Binary Trees

Performance Objectives - Students will be able to:

- Define a binary tree recursively.
- Explain terms related to binary trees.
- Describe operations that create binary trees and form left and right children to a specified node.
- Provide a formula for the number of nodes in a complete binary tree.

Topic 82 - Binary Tree Sort

Performance Objectives - Students will be able to:

- Discuss a technique to organize numerical data in a binary tree.
- Traverse a binary tree using the inorder method.
- Describe an algorithm for sorting numbers using a binary tree

Topic 83 - Binary Tree Traversal

Performance Objectives - Students will be able to:

- Traverse a binary tree in preorder, inorder and postorder.
- Form binary tree representations of algebraic expressions that contain infix operators.
- Demonstrate how traversals of binary tree expressions produce prefix, infix and postfix forms of that expression.

Topic 84 - Array Implementation of Binary Trees

Performance Objectives - Students will be able to:

- Write a declaration that implements binary tree nodes using array elements.
- Create subprograms for common tree operations using the array implementation.
- Traverse a binary tree using an array implementation.

Topic 85 - Dynamic Implementation of Binary Trees

Performance Objectives - Students will be able to:

- Declare array nodes using dynamic variables.
- Write programs that use the dynamic implementation of binary trees to create children and traverse the tree.
- Compare the array and dynamic implementation of binary trees.

Topic 86 - Huffman Algorithm

Performance Objectives - Students will be able to:

- Describe how a message can be encoded into binary bits.
- State the Huffman Algorithm that uses a binary tree to minimize the bit code for a message.
- Demonstrate how a message can be efficiently coded.

Topic 87 - Non-Binary Trees

Performance Objectives - Students will be able to:

- Define a tree and related terms.
- Describe the array and dynamic variable implementations of general trees.
- Write a subprogram that adds a child to a node of a tree.
- Describe how a general tree can be represented using a binary tree.

Topic 88 - Tree Representation of General Expressions

Performance Objectives - Students will be able to:

- Demonstrate how an ordered tree may be used to represent a general expression.
- Describe an algorithm that evaluates expressions represented by trees.
- Traverse a tree/forest in preorder, inorder and postorder.

Topic 89 - Introduction to Sorting

Performance Objectives - Students will be able to:

- Define terminology related to sorting.
- Discuss efficiency considerations that help determine the most appropriate sort in simple cases.
- Define Big-O notation.
- Compare the difference in efficiency between O(N) and O(N-squared) sorting time.

Topic 90 - Sorting - The Bubble Sort

Performance Objectives - Students will be able to:

- Discuss the efficiency considerations that help determine the most appropriate sort for a particular application.
- Explain why the bubble sort is of order N-squared.
- Show the state of the array items after each pass of the bubble sort.

Topic 91 - Binary Tree Sort

Performance Objectives - Students will be able to:

- Write a binary tree sort algorithm.
- Discuss why the relative efficiency of the binary tree sort depends on the original order of the data as well as the amount of data.
- Show that the efficiency of the binary tree sort is O(N * LOG N).
- Demonstrate how a list of items is sorted using the binary tree sort.

Topic 92 - Selection Sort

Performance Objectives - Students will be able to:

- Define a selection sort.
- Write a routine for a selection sort.
- Prove that a selection sort is O(n-sqared).
- Show the state of array items after each pass of the selection sort.

Topic 93 - Insertion Sort

Performance Objectives - Students will be able to:

. Define an insertion sort.
. Write an insertion sort algorithm.
. Prove that the insertion sort is 0(n-squared).
. Discuss how a simple insertion sort can be improved using a binary insertion or list insertion.
. Demonsrate a shell sort algortihm.

Topic 94 - Merge Sort

Performance Objectives - Students will able to:

. Show how two ordered arrays can be merged.
. Explain the code for a merge sort procedure.
. Write a merge sort procedure.
. Discuss the efficiency of a merge sort.

Topic 95 - Searching-Overview and Sequential Search

Performance Objectives - Students will be able to:

. Discuss the action of a searching algorithm.
. Discuss the relationship between searching and sorting.
. Code a sequential search of a file.

Topic 96 - Efficiency of Search Algorithms

Performance Objectives - Students will be able to:

. Tell how to measure the efficiency of a sequential search.
. Compare the efficiency of a sequential search with that of the binary search.
. Discuss techniques which can be used to improve the efficiency of a sequential search.

Topic 97 - Indexing a Table

Performance Objectives - Students will be able to:

. Describe how an index can be used to help organize a table.
. Write algorithms to construct an index for a table.
. Write algorithms to use an indexed table in searching.

Topic 98 - Searching a Binary Tree

Performance Objectives - Students will be able to:

- Demonstrate how to form a balanced binary search tree.
- Show how insertions and deletions can be made in a binary search tree.
- Discuss the efficiency of using binary search trees.

Topic 99 - Hashing

Performance Objectives - Students will be able to:

- Define a hash function and hash clashes (collisions).
- Describe how collisions can be resolved by rehashing.
- Discuss properties of a good hash function.
- Indicate methods of choosing a hash function.

Topic 100 - Chaining

Performance Objectives - Students will be able to:

- Cite problems relating to rehashing.
- Describe how collisions can be resolved by chaining.
- Explain a search and insert algorithm which implements chaining.
- Discuss advantages and disadvantages of chaining.

EVALUATION QUESTIONNAIRE

New York City Board of Education
Teacher's Guide to Advanced Placement Computer Science with Pascal - Volume I

This is an experimental bulletin in computer education. It is a pioneering effort to standardize computer instruction on a citywide basis. Your suggestions for improvement are most welcome.

1. How many classes in your school pursued this course of study? ___________

2. What were the grade levels of the students taking this course? ___________

3. Is the format of the bulletin useful? ___________________________________

4. Is the bulletin well organized? _______________________________________

5. Are the goals of the computer literacy course clearly stated?

   ____

6. Are the activities appropriate for the intended grade levels?

   ____

7. Is there sufficient emphasis on enduring computer concepts, such as good documentation, structured programming, modular design, problem-solving skills, and so forth? ______________________________________

   ____

8. Did the learning activities enable students to achieve the performance objectives? ____

   ____

9. Was there a significant variety of activities in the learning strategies?

   ____

   ____

10. What changes (if any) would you make in the learning activities presented in the bulletin? ____

    ____

11. What additions would you make to the scope of the course content?

    ____

    ____

12. What topics in the course of study would you delete or change?

____

____

13. Was the interest level of the materials high among the students?

____

14. What changes in sequence would you recommend? ____________________

____

15. What difficulty (if any) did you encounter in implementing this course?

____

____

16. What do you consider to be the overall strengths of the bulletin? (If any)

____

____

17. What do you consider to be the overall weaknesses of the bulletin? (If any)

____

____

18. What changes would you suggest to improve the quality of the bulletin? (If any)

____

____

19. Please make any additional comments you feel are appropriate. __________

____

____

____

Return to: Mr. Marc Licht, Director
Computer and Information Sciences Unit
131 Livingston Street - Room 409
Brooklyn, N.Y. 11201